Lecture Notes in Electrical Engineering 361

Frank Oppenheimer
Julio Luis Medina Pasaje
Editors

Languages, Design Methods, and Tools for Electronic System Design

Selected Contributions from FDL 2014

Springer

Lecture Notes in Electrical Engineering

Volume 361

Board of Series editors

Leopoldo Angrisani, Napoli, Italy
Marco Arteaga, Coyoacán, México
Samarjit Chakraborty, München, Germany
Jiming Chen, Hangzhou, P.R. China
Tan Kay Chen, Singapore, Singapore
Rüdiger Dillmann, Karlsruhe, Germany
Haibin Duan, Beijing, China
Gianluigi Ferrari, Parma, Italy
Manuel Ferre, Madrid, Spain
Sandra Hirche, München, Germany
Faryar Jabbari, Irvine, CA, USA
Janusz Kacprzyk, Warsaw, Poland
Alaa Khamis, New Cairo City, Egypt
Torsten Kroeger, Stanford, CA, USA
Tan Cher Ming, Singapore, Singapore
Wolfgang Minker, Ulm, Germany
Pradeep Misra, Dayton, OH, USA
Sebastian Möller, Berlin, Germany
Subhas Mukhopadyay, Palmerston, New Zealand
Cun-Zheng Ning, Tempe, AZ, USA
Toyoaki Nishida, Kyoto, Japan
Bijaya Ketan Panigrahi, New Delhi, India
Federica Pascucci, Roma, Italy
Tariq Samad, Minneapolis, MN, USA
Gan Woon Seng, Singapore, Singapore
Germano Veiga, Porto, Portugal
Haitao Wu, Beijing, China
Junjie James Zhang, Charlotte, NC, USA

About this Series

"Lecture Notes in Electrical Engineering (LNEE)" is a book series which reports the latest research and developments in Electrical Engineering, namely:

- Communication, Networks, and Information Theory
- Computer Engineering
- Signal, Image, Speech and Information Processing
- Circuits and Systems
- Bioengineering

LNEE publishes authored monographs and contributed volumes which present cutting edge research information as well as new perspectives on classical fields, while maintaining Springer's high standards of academic excellence. Also considered for publication are lecture materials, proceedings, and other related materials of exceptionally high quality and interest. The subject matter should be original and timely, reporting the latest research and developments in all areas of electrical engineering.

The audience for the books in LNEE consists of advanced level students, researchers, and industry professionals working at the forefront of their fields. Much like Springer's other Lecture Notes series, LNEE will be distributed through Springer's print and electronic publishing channels.

More information about this series at http://www.springer.com/series/7818

Frank Oppenheimer • Julio Luis Medina Pasaje
Editors

Languages, Design Methods, and Tools for Electronic System Design

Selected Contributions from FDL 2014

Springer

Editors
Frank Oppenheimer
Transportation Division
OFFIS e.V., Oldenburg, Germany

Julio Luis Medina Pasaje
Universidad de Cantabria
Santander, Spain

ISSN 1876-1100 ISSN 1876-1119 (electronic)
Lecture Notes in Electrical Engineering
ISBN 978-3-319-24455-6 ISBN 978-3-319-24457-0 (eBook)
DOI 10.1007/978-3-319-24457-0

Library of Congress Control Number: 2015957253

Springer Cham Heidelberg New York Dordrecht London
© Springer International Publishing Switzerland 2016
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made.

Printed on acid-free paper

Springer International Publishing AG Switzerland is part of Springer Science+Business Media (www.springer.com)

Contents

Part I
Formal Models and Verification and Predictability

Chapter 1
Automatic Refinement Checking for Formal System Models

Julia Seiter, Robert Wille, Ulrich Kühne, and Rolf Drechsler

Abstract For the design of complex systems, formal modelling languages such as UML or SysML find significant attention. The typical model-driven design flow assumes thereby an initial (abstract) model which is iteratively refined to a more precise description. During this process, new errors and inconsistencies might be introduced. In this chapter, we propose an automatic method for verifying the consistency of refinements in UML or SysML. For this purpose, a theoretical foundation is considered from which the corresponding proof obligations are determined. Afterwards, they are encoded as an instance of *satisfiability modulo theories (SMT)* and solved using proper solving engines. The practical use of the proposed method is demonstrated and compared to a previously proposed approach.

1.1 Introduction

Due to the increasing complexity of today's systems and, caused by this, the steady strive of designers and researchers towards higher levels of abstractions, modelling languages such as the *unified modeling language* (UML) [28] or the *Systems Modeling Language* (SysML) [33] together with textual constraints, e.g., provided by the *object constraint language* (OCL) [32] received significant attention in computer-aided design. They allow for a *formal* specification of a system prior to the implementation. Such an initial blueprint can be iteratively refined to a final model to be implemented. The actual implementation is then carried out manually, by using automatic code generation, or a mix of both.

An advantage of using formal descriptions like UML or SysML is that the initial system models can already be subject to (automatic) correctness and plausibility

J. Seiter (✉) • U. Kühne
Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
e-mail: jseiter@informatik.uni-bremen.de; ulrichk@informatik.uni-bremen.de

R. Wille • R. Drechsler
Institute of Computer Science, University of Bremen and Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
e-mail: rwille@informatik.uni-bremen.de; drechsle@informatik.uni-bremen.de

© Springer International Publishing Switzerland 2016

F. Oppenheimer, J.L. Medina Pasaje (eds.), *Languages, Design Methods, and Tools for Electronic System Design*, Lecture Notes in Electrical Engineering 361,
DOI 10.1007/978-3-319-24457-0_1

checks. By this, inconsistencies and/or errors in the specification can be detected even before a single line of code has been written. For this purpose, several approaches have been introduced [3, 11, 12, 30, 31]. They tackle verification questions such as *"Does the conjunction of all constraints still allow the instantiation of a legal system state?"* or *"Is it possible to reach certain bad states, good states, or deadlocks?"*. These verification tasks are typically categorized in terms such as consistency, reachability, or independence [17].

However, these verification techniques are usually carried out on a single model and, hence, are not sufficient for the typical model-driven design flow in which an abstract model is generated first and iteratively refined to a more precise description. Indeed, they enable the detection of errors and inconsistencies in one iteration, but they need to be re-applied in the succeeding iteration even for minor changes. Instead, it is desirable to check whether a refined model is still consistent to the original abstract model. In this way, verification results from abstract models will also be valid for later refined models.

For the creation of software systems, such a refinement process has already been established. Here, frameworks such as the *B-method* [1], *Event-B* [2], and *Z* [34] exist. These methods rely on a rigorous modelling using first-order logic. Extensions, e.g., of *Event-B* to the UML-like *UML-B* or translations of UML to *B* models, are available in [5, 29], respectively. But since the proof obligations for a correct refinement in these frameworks are undecidable in general, usually manual or interactive proofs must be conducted—a time-consuming process which additionally requires a thorough mathematical background.

Hence, *automatic* proof techniques are desired. For this purpose, existing solutions proposed in the context of hardware verification and the design and modelling of reactive systems may be applied. Here, the relation of an implementation and its specification (comparable to a refined and an abstract system) is traditionally described by *simulation relations* on finite state systems (see, e.g., [7, 16, 21, 23]). There exist algorithms for computing such relations [10, 25]. However, since these algorithms operate on explicit state graphs, they do require the consideration of all possible system states and operation calls—infeasible for larger designs. A similar difficulty occurs when attempting to automatize the verification process proposed by the B-method. In [20], an extension to the tool ProB has been proposed which automatically solves all proof obligations created in the refinement process. However, according to their evaluation, the run-time for the verification grows exponentially.

As a consequence, an alternative solution is proposed in this chapter which exploits the recent accomplishments in the domain of model-based verification (i.e. approaches like [3, 11, 12, 30, 31]) using a symbolic state representation. Based on a theoretical foundation of refinement, we can prove the preservation of safety properties from an abstract model to a more detailed model. In contrast to the existing approaches like in [24], this also includes *non-atomic* refinements, where an abstract operation is replaced by a sequence of refined operations. By this, the consistency of a refined model against an original (abstract) model can be checked automatically.

The remainder of the chapter is structured as follows. The chapter starts with a brief review on models and their notation in Sect. 1.2. Section 1.3 describes the addressed problem which, afterwards, is formalized in Sect. 1.4. The proposed solution is introduced in Sect. 1.5 and its usefulness is demonstrated in Sect. 1.6 where it is applied to several examples and compared to the results from [20]. The chapter is concluded in Sect. 1.8.

1.2 Models and Their Notation

Modelling languages provide a description of a system to be realized, i.e. proper description means to *formally* define the structure and the behaviour of a system. At the same time, implementation details which are not of interest in the early design/specification state remain hidden. In the following, we briefly review the respective description by means of UML and OCL. The approaches proposed in this chapter can be applied to similar modelling languages (e.g. such as SysML) as well.

Definition 1. A *model* is a tuple $m = (C, R)$ composed of a set of classes C and a set of associations R. A *class* $c = (A, O, I) \in C$ of a model m is a tuple composed of attributes A, operations O, and invariants I. An n-ary *association* $r \in R$ of a model m is a tuple $r = (r_{\text{ends}}, r_{\text{mult}})$ with *association ends* $r_{\text{ends}} \in C^n$ for a given set of classes C and *multiplicities* $r_{\text{mult}} \in (\mathbb{N}_0 \times \mathbb{N})^n$ that is defined as a range with a *lower bound* and an *upper bound*.

Example 1. Figure 1.1a shows a model composed of the class Phone which itself is composed of the attributes $A = \{\text{credit}\}$, the operations $O = \{\text{charge}\}$, and the invariant $I = \{\text{i1}\}$.

Invariants in the model describe additional constraints which have to be satisfied by each instantiation of the model. For this purpose, textual descriptions provided in OCL can be applied. OCL also allows the specification of the behaviour of operations.

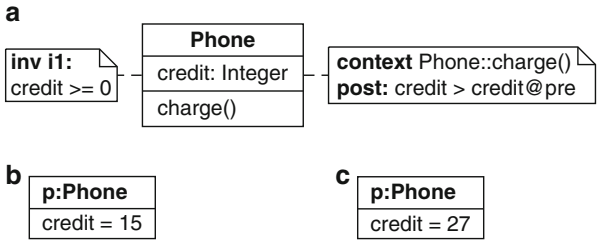


Fig. 1.1 Example of a model and its instantiation. (**a**) Given model; (**b**) State σ_0; (**c**) State σ_1

Definition 2. *OCL expressions* Φ are textual constraints over a set of *variables* $V \supseteq A \times R$ composed of the attributes A of the respective classes, but also further (auxiliary) variables. An OCL condition $\varphi \in \Phi$ is defined as a function $\varphi : V \to \mathbb{B}$. They can be applied to specify the invariants of a class as well as the pre- and post-condition of an operation, i.e. $I \subseteq \Phi$. An *operation* $o \in O$ is defined as a tuple $o = (\lhd, \rhd)$ with pre-condition $\lhd \in \Phi$ and post-condition $\rhd \in \Phi$, respectively. The valid initial assignments of a class are described by a predicate init $\in \Phi$.

Example 2. In the model from Fig. 1.1a, the invariant i1 states that credit always has to be greater or equal to 0. The post-condition of the operation charge ensures that after invoking the operation, credit is increased.

Any instance of a model is called a *system state* and is visualized by an object diagram.

Definition 3. *Object diagrams* represent precise system states in a model. A *system state* is denoted by σ and is composed of *objects*, i.e. instantiations of classes. The attributes of the objects are derived from the classes and assigned precise values. Associations are instantiated as precise *links* between objects.

In order to evaluate a model, it is crucial to particularly consider whether system states are valid or sequences of system states represent valid behaviour. This requires the evaluation of the given OCL expressions.

Definition 4. For a system state σ and an OCL expression φ, the evaluation of φ in σ is denoted by $\varphi(\sigma)$. A system state σ for a model $m = (C, R)$ is called *valid* iff it satisfies all invariants of m, i.e. iff $\bigwedge_{c \in C} I_c(\sigma)$. An operation call is valid iff it transforms a system state σ_t satisfying the pre-condition to a succeeding system state σ_{t+1} satisfying the post-condition,[1] i.e. iff $\lhd(\sigma_t)$ and $\rhd(\sigma_t, \sigma_{t+1})$. A sequence of system states is called valid, if all operation calls are valid.

Example 3. Figures 1.1b and c show two valid system states (in terms of object diagrams) for the model from Fig. 1.1a. This is a valid sequence of system states which can be created by calling the operation charge.

1.3 Refinement of Models

Using the description means reviewed in the previous section allows for a *formal* specification of a system to be implemented. By this, precise blueprints are available already in the early stages of the design. A rough initial model is thereby created

[1] The post-condition is a binary predicate, since it can also depend on the source state, which is expressed using @pre in OCL.

first which covers the most important core functionality. Afterwards, a *refinement* process is conducted in which a more precise model of the respective components and operations is created. This refinement process may include

- the addition of new components and relations (i.e. classes and their associations),
- the extension of classes by new attributes,
- the extension of the behavioural description (i.e. the addition of new operations as well as pre- and post-conditions and the strengthening of existing pre- and post-conditions), and
- the extension of the constraints (i.e. the addition of new and the strengthening of existing invariants).

Example 4. Consider the model from Fig. 1.2a representing a simple phone application. It consists of a phone with a credit which can be charged by a corresponding operation. A possible refinement of this model is depicted in Fig. 1.2b. Here, the post-condition of the operation *charge* has been rendered more precise, i.e. a parameter defining the amount of credits to be charged has been added.

Remark. Up to this point, we do not consider the refinement of associations and operation parameters. This includes the type of association and the multiplicities of the associations ends. However, this is not due to a technical limitation of our approach which can easily be extended to further description means. Here, we decided to focus on the refinement of attributes and operations, considering in particular non-atomic refinement, as these are the most important modelling elements in formal system specifications. Other kinds of refinement, e.g. for operation parameters, can be conducted analogously.

In the following, we denote the *abstract model* by m^a and the *refined model* by m^r. A refinement is described by a refinement relation defined as follows:

Definition 5. A refinement relation is a pair $\text{Ref} = (\text{Ref}_\Sigma, \text{Ref}_\Omega)$ with

- Ref_Σ describing the refinement of the states, i.e. Ref_Σ^{-1} is a function mapping a refined state σ^r to its corresponding abstract state σ^a, and
- Ref_Ω describing the refinement of operations, i.e. Ref_Ω is a function mapping an abstract operation o^a to a sequence $o_1^r \cdot o_2^r \cdot \cdots \cdot o_k^r \in (O^r)^+$ of refined operations.

Example 5. The refinement from the model in Fig. 1.2a to the model in Fig. 1.2b is described by the relation $\text{Ref}=(\text{Ref}_\Sigma, \text{Ref}_\Omega)$. That is, each state σ^r in the

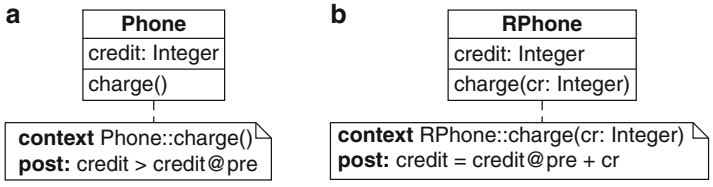


Fig. 1.2 Refinement step. (**a**) Abstract model; (**b**) Refined model

refined model (composed of objects from class RPhone) has one corresponding state $\text{Ref}_{\Sigma}^{-1}(\sigma^r) = \sigma^a$ in the abstract model (composed of objects from class Phone) such that RPhone.credit = Phone.credit. Furthermore, the operation Phone::charge() is refined so that Ref_{Ω}(Phone::charge()) = RPhone::charge(cr), i.e. a corresponding operation with an additional parameter.

Adding details step by step—like in the above example—is common practice in model-driven design using UML or SysML. Nevertheless, during this manual process, new errors might be introduced, leading to a refined model that is not consistent with the abstract model any more. In fact, the refinement sketched above contains a serious flaw.

Example 6. The refined model in Fig. 1.2b allows for a behaviour that is not specified by the abstract model. It is possible to assign a value equal to or less than 0 to the parameter cr, so that after calling the operation charge, the value of credit does not change at all or even decreases. This contradicts the behaviour described in the abstract model which only allows for a strict increase of that attribute. As a possible repair of this inconsistency, the precondition pre: cr > 0 could be added to the operation RPhone::charge(cr).

In order to identify and fix inconsistencies of the refinement, designers have to intensely check the refined model against the abstract original—often a complicated and cumbersome task which results in a manual and time-consuming procedure. In the worst case, all components, constraints, and possible executions have to be inspected. While this might be feasible for the simple model discussed above, it becomes highly inefficient for larger models. Hence, in the remainder of this chapter we consider the question "How to automatically check whether a refined model m^r is consistent with respect to the originally given abstract model m^a?"

1.4 Theoretical Foundation

This section formalizes the problem sketched above. For this purpose, we exploit the theoretical foundation of Kripke structures and their concepts of simulation relations. We show how these concepts can be applied for the refinement of system models provided e.g. in UML or SysML. This provides the basis for the proposed solution which is described afterwards in Sect. 1.5.

Since we are considering models mostly in the context of software and hardware systems, we assume bounded data types and a bounded number of instances in the following.[2] Based on these assumptions, the behaviour of a model can be described as a finite state machine, e.g. a *Kripke structure*.

[2]This restriction is common in many approaches (e.g. [3, 11, 12, 30, 31]) and also justified by the fact that, eventually, the implemented system will be realized by bounded physical devices anyway.

Definition 6. A Kripke structure is a tuple $\mathcal{K} = (S, S_0, AP, \mathcal{L}, \rightarrow)$ with a finite set of states S, initial states $S_0 \subset S$, a set of atomic propositions AP, a labelling function $\mathcal{L} : S \rightarrow 2^{|AP|}$, and a (left-total) transition relation $\rightarrow \subseteq S \times S$.

Using this formalism, we can define the behaviour of a UML or SysML model and its operations as follows:

Definition 7. A model $m = (C, R)$ induces a Kripke structure $\mathcal{K}_m = (S, S_0, AP, \mathcal{L}, \rightarrow)$ with

- S being the set of all valid system states of $m = (C, R)$,
- S_0 being the set of initial states defined by the predicate init (cf. Definition 2), i.e. $S_0 = \{\sigma \in S \mid \text{init}(\sigma)\}$,
- $\rightarrow$ being the transition relation including the identity (i.e. $\sigma \rightarrow \sigma$) as well as all transitions caused by executing operations $o = (\triangleleft, \triangleright) \in O$ of the model (i.e. $\sigma_1 \rightarrow \sigma_2$ with $\triangleleft(\sigma_1)$ and $\triangleright(\sigma_1, \sigma_2)$), and
- AP and $\mathcal{L}$ are defined s.t. $\mathcal{L}$ can be used to retrieve the values of the attributes of σ in the usual bit-vector encoding.

We will write $\sigma_1 \overset{o}{\rightarrow} \sigma_2$ to make clear that an operation o transforms a state σ_1 to a state σ_2.

With this formalization, we can make use of known results for finite and reactive systems. To describe refinements in this domain, *simulation relations* are usually applied for this purpose (see, e.g., [7, 10, 16, 21, 23, 25]). In this chapter, we adapt this concept for the considered formal models. This leads to the following definition of a simulation relation.

Definition 8. Let $\mathscr{A} = (S_\mathscr{A}, S_{\mathscr{A}0}, AP_\mathscr{A}, \mathcal{L}_\mathscr{A}, \rightarrow_\mathscr{A})$ be a Kripke structure of an abstract model and $\mathscr{R} = (S_\mathscr{R}, S_{\mathscr{R}0}, AP_\mathscr{R}, \mathcal{L}_\mathscr{R}, \rightarrow_\mathscr{R})$ be a Kripke structure of a refined model with $AP_\mathscr{R} \supseteq AP_\mathscr{A}$. Then, a relation $H \subseteq S_\mathscr{R} \times S_\mathscr{A}$ is a simulation relation iff

1. all initial states in the refined model have a corresponding initial state in the abstract model, i.e. $\forall s_0 \in S_{\mathscr{R}0} \exists s'_0 \in S_{\mathscr{A}0}$ with $H(s_0, s'_0)$,
2. all states in the refined model are constrained by at least the same propositions as their corresponding abstract state, i.e. $\forall s, s' : H(s, s') \Rightarrow \mathcal{L}_\mathscr{R}(s) \cap AP_\mathscr{A} = \mathcal{L}_\mathscr{A}(s')$, and
3. all possible transitions in the refined model have a corresponding transition in the abstract model leading to a corresponding succeeding state, i.e. $\forall s, s' : H(s, s') \Rightarrow s \rightarrow_\mathscr{R} t \Rightarrow \exists t' \in S_\mathscr{A}$ s.t. $s' \rightarrow_\mathscr{A} t'$ and $H(t, t')$.

We say that $\mathscr{R}$ is simulated by $\mathscr{A}$ (written as $\mathscr{R} \preceq \mathscr{A}$), if there exists a simulation relation.

Example 7. As an illustration of the above definition, Fig. 1.3a shows the general scheme of a transition between states from a refined model (denoted by s and t) and a corresponding transition in an abstract model (from s' to t'). The simulation relation H is indicated by dashed lines. Figure 1.3b on the right shows an example

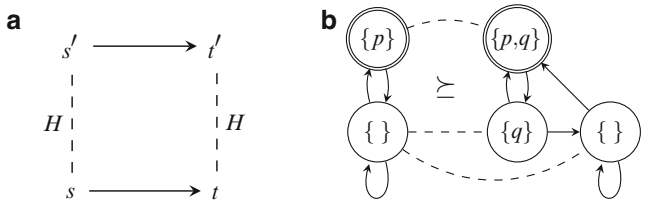


Fig. 1.3 Simulation relation. (**a**) Correspondence of states; (**b**) Example for simulation

for two Kripke structures. The abstract model is the one on the left-hand side and simulates the refined model on the right-hand side. Initial states are marked by a double outline. While all corresponding states agree on the atomic proposition p, the refined model has an additional proposition q. It can easily be checked that for each refined transition, there is a corresponding abstract one.

The simulation relation ensures that a refined model is consistent to an abstract system, i.e. whatever the refined system does must be allowed by the abstract system. Besides that, there might be more behaviour allowed in the abstract system than implemented. If we have $\mathscr{R} \preceq \mathscr{A}$, then the traces of $\mathscr{R}$ are contained in those of $\mathscr{A}$. This also means that globally valid properties of $\mathscr{A}$ carry over to $\mathscr{R}$, as, for example, the non-reachability of bad states. Hence, by proving that the applied refinement Ref (cf. Definition 5) satisfies the properties of a simulation relation H, the consistency of a refined model can be verified.

However, determining a simulation relation requires a strict step-wise correspondence between the transition in the refined model and in the abstract one. But refinements of UML or SysML models often include the replacement of a single abstract operation by a sequence of refined operations (also known as *non-atomic refinement* [6]). In order to formalize this, we need a more flexible relation. This is provided by the notion of *divergence-blind stuttering simulation* (dbs-simulation).

Definition 9. Given two Kripke structures $\mathscr{R}$ and $\mathscr{A}$ with $AP_{\mathscr{R}} \supseteq AP_{\mathscr{A}}$, a relation $H \subseteq S_{\mathscr{R}} \times S_{\mathscr{A}}$ is a divergence blind stuttering simulation (dbs-simulation) iff

1. $\forall s_0 \in S_{\mathscr{R}0} \exists s'_0 \in S_{\mathscr{A}0}$ with $H(s_0, s'_0)$,
2. $\forall s, s' : H(s, s') \Rightarrow \mathscr{L}_{\mathscr{R}}(s) \cap AP_{\mathscr{A}} = \mathscr{L}_{\mathscr{A}}(s')$, and
3. each possible transition in the refined model corresponds to a sequence of 0 or more abstract transitions, i.e. $\forall s, s' : H(s, s')$ and $s \rightarrow_{\mathscr{R}} t$, then there exist $t'_0, t'_1 \ldots t'_n$ $(n \geq 0)$ such that $s' = t'_0$ and $\forall i < n : t'_i \rightarrow_{\mathscr{A}} t'_{i+1} \wedge H(s, t'_i)$ and $H(s', t'_n)$.

We say that $\mathscr{R}$ is dbs-simulated by $\mathscr{A}$, written as $\mathscr{R} \preceq_{\mathsf{dbs}} \mathscr{A}$, if there exists a dbs-simulation.

Compared to the original simulation relation, this definition is less precise with respect to the *duration* of specific operations. But, it still guarantees that the functional behaviour of the refined model is consistent with the behaviour of the

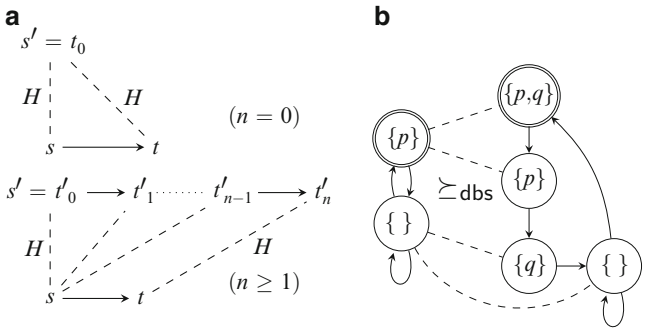


Fig. 1.4 dbs—simulation relation. (**a**) Correspondence of states; (**b**) Example for dbs-simulation

abstract model—even in the absence of a (step-wise) one-to-one correspondence of the transitions. In particular, if the properties of the dbs-simulation are satisfied, a bad state unreachable in $\mathscr{A}$ is also unreachable in $\mathscr{R}$.

Example 8. In Fig. 1.4, the dbs-simulation relation is illustrated. The general scheme of corresponding states and transitions is shown in Fig. 1.4a. In Fig. 1.4b, the abstract model on the left-hand side dbs-simulates the refined model on the right-hand side. Note that the transition from the initial state of the refined model is a *stuttering* transition, since it corresponds to an empty sequence of transitions in the abstract model.

The above definitions provide the formal foundation for consistency checks of refinements. By referring to dbs-simulation, we can preserve safety properties from an abstract model to a refined model. Hence, by proving that the actually applied refinement Ref (c.f. Definition 5) indeed satisfies the properties of a dbs-simulation H (cf. Definition 9), the consistency of the refinement is shown. In the next section, we describe how the refinement for UML or SysML models can efficiently be checked.

1.5 Proposed Solution

In this section, we present the proposed solution to automatically check the refinement of the model m^a to the model m^r. As outlined above, we particularly require that the applied refinement Ref satisfies the properties of a dbs-simulation H. For this purpose, *all* (valid) system states as well as *all* possible operation calls in those states need to be considered. Naive schemes, e.g., relying on enumerating all possible scenarios are clearly infeasible for this purpose. Hence, we propose an approach that maps the problem to an instance of *satisfiability modulo theories* (SMT) and, afterwards, exploits the efficiency of corresponding solving techniques (such as [9]).

To this end, we represent arbitrary system states and transitions for the abstract model m^a as well as the refined model m^r together with their invariants and the refinement relation Ref in terms of bit-vectors and bit-vector constraints. In the same way, the verification objectives proving that the applied refinement Ref indeed ensures dbs-simulation are encoded and checked automatically. In the following, the resulting verification objectives are briefly sketched. Then, we illustrate how to encode these in SMT.

1.5.1 Verification Objectives

As motivated in Sect. 1.3, we are interested in the relation between abstract operations and their possibly non-atomic refinements. These operation refinements are given as operation sequences according to Definition 5. By this, the refinement check is reduced to the question of whether there is a sequence of operation calls in the refined model that corresponds to a single call in the abstract model (according to the given refinement relation), but violates the requirements of the abstract operation. Unsatisfiability of such an instance shows that no such sequence exists and, hence, the refinement is correct. Otherwise, a counterexample showing the inconsistency is provided.

Based on this intuitive notion of refinement, we derive three verification objectives that prove the correspondence of an abstract operation and its refined operations and are sufficient to prove dbs-simulation. By this, the preservation of safety properties is guaranteed and the refinement is proven consistent. The three objectives read as follows:

1. Check whether all initial states in the refined model indeed correspond to the respective initial states in the abstract model, i.e.

$$\forall \sigma_0^r : \text{init}(\sigma_0^r) \Rightarrow \text{init}(\text{Ref}_\Sigma^{-1}(\sigma_0^r)).$$

   This check is illustrated in Fig. 1.5a.
2. For each step o_j^r of the refined operation which transforms a refined state σ_1^r, check whether this step does not lead to a succeeding state o_2^r which is inconsistent to its corresponding abstract states. In fact, the succeeding state σ_2^r either has to correspond to the unchanged abstract state or to its abstract state which results after applying the corresponding abstract operation o^a, i.e. for each step o_j^r

$$\begin{aligned}\forall \sigma^a, \sigma_1^r, \sigma_2^r : &\text{Ref}_\Sigma(\sigma^a, \sigma_1^r) \wedge \sigma_1^r \xrightarrow{o_j^r} \sigma_2^r \\ &\Rightarrow \left(\text{Ref}_\Sigma^{-1}(\sigma_2^r) = \sigma^a \vee (\lhd_{o^a}(\sigma^a) \wedge \rhd_{o^a}(\sigma^a, \text{Ref}_\Sigma^{-1}(\sigma_2^r)))\right)\end{aligned}$$

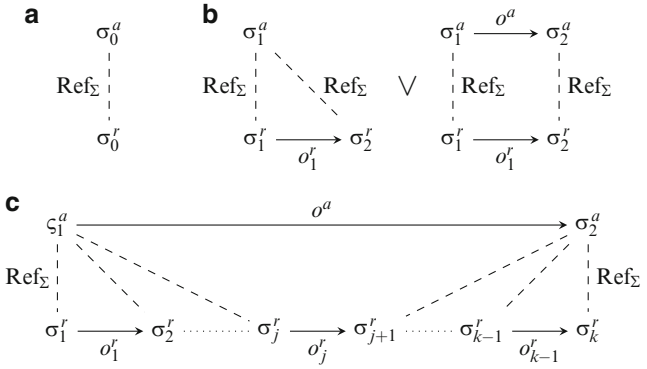


Fig. 1.5 Verification objectives. (**a**) Initialization; (**b**) Single step correspondence; (**c**) Chaining of refined steps

is checked. This check is illustrated in Fig. 1.5b. These two objectives are already sufficient to prove dbs-simulation. Nevertheless, a third objective is additionally checked.

3. Check whether the joint effect of the refined operation sequence adheres exactly to the specification of the abstract operation. That is, for each operation o^a and its refinement $o^r_1 \dots o^r_k$

$$\forall \sigma^a_1, \sigma^r_1, \sigma^r_2 \dots \sigma^r_{k+1} : \mathrm{Ref}_\Sigma(\sigma^a_1, \sigma^r_1) \wedge \sigma^r_1 \xrightarrow{o^r_1} \sigma^r_2 \dots \sigma^r_k \xrightarrow{o^r_k} \sigma^r_{k+1}$$
$$\Rightarrow \left(\lhd_{o^a}(\sigma^a_1) \wedge \rhd_{o^a}(\sigma^a_1, \mathrm{Ref}^{-1}_\Sigma(\sigma^r_{k+1})) \right)$$

is checked. This check is illustrated in Fig. 1.5c. This check particularly considers the common UML or SysML refinement which often refines a single abstract operation into a sequence of refined operations.

Together, these three objectives represent the verification tasks to be solved by the respective solving engine. Next, we illustrate how they are encoded as an SMT instance.

1.5.2 Basic Encoding

In order to represent arbitrary system states and transitions in an SMT instance, we use an encoding similar to the ones previously presented, e.g., in [3, 11, 12, 30] and particularly in [31]. Here, systems states (basically defined by the values of their attributes) and links are represented by corresponding bit-vector variables. Invariants are represented by corresponding SMT constraints. By this, it is ensured

that the solving engine only considers systems states σ composed of objects satisfying all invariants of the underlying class, i.e. $I_c(\sigma)$.

In order to encode transitions caused by operation calls, bit vectors $\boldsymbol{\omega}_i \in \mathbb{B}^{\lceil \mathrm{ld}(|O|)\rceil}$ are created for each step i in the refined model. Depending on the assignment to $\boldsymbol{\omega}$, the respective pre-conditions and post-conditions have to be enforced. This can be realized by a constraint

$$\boldsymbol{\omega}_i = \mathrm{enc}(o) \Rightarrow \lhd_o(\sigma_i^r) \wedge \rhd_o(\sigma_i^r, \sigma_{i+1}^r),$$

where $\mathrm{enc}(o)$ represents a unique binary representation of the operation o, i.e. a number from 0 to $|O_r|$ with $\mathrm{enc}(\mathrm{id}) = 0$. Furthermore, to ensure that only legal values can be assigned to a vector $\boldsymbol{\omega}$, we use a constraint $\boldsymbol{\omega} \leq |O_r|$.

We further introduce auxiliary predicates that reflect the relationship between an abstract operation and its refined steps. For this purpose, the operation refinement Ref_Ω is utilized:

$$\mathrm{step}_i(o^a, o_j^r) \Leftrightarrow \mathrm{Ref}_\Omega(o^a) = o_1 \cdot o_2 \ldots o_k \wedge o_i = o_j$$

$$\mathrm{step}(o^a, o_j^r) \Leftrightarrow \bigvee_{i=1}^{|\mathrm{Ref}_\Omega(o^a)|} \mathrm{step}_i(o^a, o_j^r).$$

Here, $\mathrm{step}_i(o^a, o_j^r)$ evaluates to true iff the refined operation o_j^r is the ith step in the refinement of o^a, while $\mathrm{step}(o^a, o_j^r)$ reflects that o_j^r occurs in any position in the refinement of o^a.

In order to encode the chaining of the refined operation steps according to the scheme in Fig. 1.5c, we define the predicate chain:

$$\mathrm{chain}(o^a) \Leftrightarrow \bigwedge_{i=1}^{l} \big(\mathrm{step}_i(o^a, o_i^r) \wedge \boldsymbol{\omega}_i = \mathrm{enc}(o_i^r) \vee i > |\mathrm{Ref}_\Omega(o^a)| \wedge \boldsymbol{\omega}_i = \mathrm{enc}(\mathrm{id})\big).$$

In the above formula, in order to cover all abstract operations in one instance, the refined operation sequences are brought to the same maximal length l by filling up the sequence with the identity function for operations where $|\mathrm{Ref}_\Omega(o)| < l$. We thereby make use of the maximum number of steps according to Ref_Ω, i.e. $l = \max\{|\mathrm{Ref}_\Omega(o^a)| \mid o^a \in O^a\}$. Next, the above "ingredients" are put together in order to encode the verification objectives of a refinement.

1.5.3 Encoding the Verification Objectives

While the encodings from above ensure a proper representation of the models, system states, and execution of operations in an SMT instance, finally the verification objectives from Sect. 1.5.1 are encoded. In order to prove (1), we encode its negation

and check for unsatisfiability, i.e.

$$\exists \sigma_0^r, \sigma_0^a : \mathrm{Ref}_\Sigma(\sigma_0^a, \sigma_0^r) \wedge \mathrm{init}(\sigma_0^r) \wedge \neg\, \mathrm{init}(\sigma_0^a). \tag{1.1}$$

To check (2), we try to determine a refined operation call that cannot be matched with one of the schemes in Fig. 1.5b. Hence, instead of encoding (2) for each individual refined operation, we let the solving engine choose a refined step that violates the requirements, i.e.

$$\begin{aligned}\exists \sigma_1^a, \sigma_2^a, \sigma_1^r, \sigma_2^r, o^a, o^r : &\mathrm{Ref}_\Sigma(\sigma_1^a, \sigma_1^r) \wedge \boldsymbol{\omega}_1 = \mathrm{enc}(o^r) \\ &\wedge \mathrm{step}(o^a, o^r) \wedge \mathrm{Ref}_\Sigma(\sigma_2^a, \sigma_2^r) \\ &\wedge \neg \left(\sigma_1^a = \sigma_2^a \vee \lhd_{o^a}(\sigma_1^a) \wedge \rhd_{o^a}(\sigma_1^a, \sigma_2^a)\right).\end{aligned} \tag{1.2}$$

That is, we check that, given a pair of corresponding states and an operation call in the refined state, whether it is possible that the reached refined state neither corresponds to the original abstract state nor does it satisfy the specification of the abstract operation. In case this instance is unsatisfiable, objective (2) has been proven.

Finally, for (3) we need to check whether we can determine an instantiated sequence of refined operation calls, such that their joint effect does not adhere to the specification of the respective abstract operation. For this purpose, we use the chain predicate as defined in the previous section to construct the *unrolled* operation sequence, i.e.

$$\begin{aligned}\exists \sigma_1^a, \sigma_2^a, \sigma_1^r \ldots \sigma_{l+1}^r, o^a, o_1^r \ldots o_l^r : &\mathrm{Ref}_\Sigma(\sigma_1^a, \sigma_1^r) \wedge \mathrm{chain}(o^a) \wedge \mathrm{Ref}_\Sigma(\sigma_1^a, \sigma_{l+1}^r) \\ &\wedge \neg \left(\lhd_{o^a}(\sigma_1^a) \wedge \rhd_{o^a}(\sigma_1^a, \sigma_{l+1}^a)\right).\end{aligned} \tag{1.3}$$

That is, we are searching for a chain of $l + 1$ refined states and connected by l operation calls such that there are no corresponding abstract states which satisfy the pre- and post-conditions of the respective abstract operation. Unsatisfiability proves that no such chain exists and, hence, objective (3) holds.

1.6 Evaluation

The approach presented in this chapter has been implemented in Java, using the SMT solver *Boolector* [9] as underlying solving engine. In order to evaluate the applicability and scalability of our approach, we have applied it to two systems based on examples presented in [1]. For the sake of comparison, these examples have additionally been verified using the previously proposed B method following a manual as well as an automatic scheme [20].

The first example describes an access control system (AC) which is employed to grant access to a building when presented with an authorized ID by a user. Two refinement steps have been modelled, a correct and an erroneous one, which are depicted in Fig. 1.6 together with the abstract model. All types of refinement

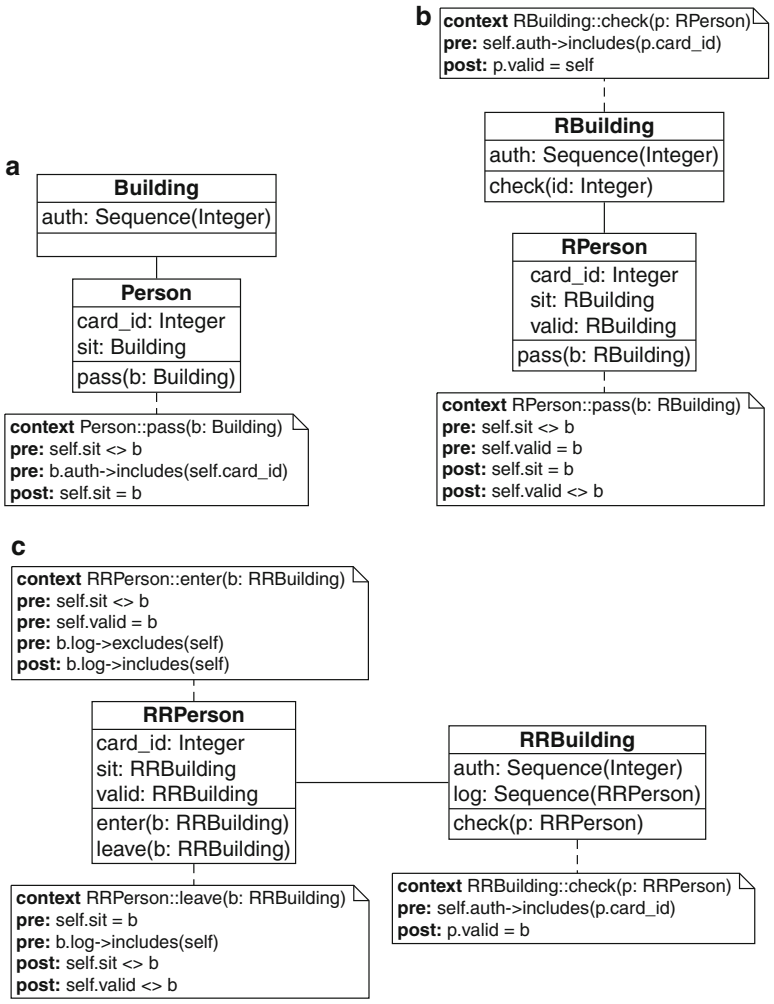


Fig. 1.6 Access control system. (**a**) Abstract model; (**b**) First refinement; (**c**) Second refinement

presented in this chapter have been applied to this model, i.e. attribute refinement as well as atomic and non-atomic operation refinement.

Table 1.1 provides the sizes of the three models (denoted by AC0, AC1, and AC2), i.e. the number of classes, attributes, operations, and OCL constraints are listed. As can be seen, the abstract model (AC0) and the two refined models (AC1, AC2) are relatively small regarding the number of UML elements. Only the number of OCL constraints increases slightly as the added and refined operations are extended.

In order to compare our work to the traditional B approach, we re-modelled this example in B and verified the refinement manually, using the event-B tool

Table 1.1 Size of examples

Model	#Classes	#Attributes	#Operations	#Constraints
AC0	2	3	1	3
AC1	2	4	2	6
AC2	2	5	3	10
MPC0	2	2	4	12
MPC1	2	6	12	40
MPC2	2	8	16	52
MPC3	2	8	16	60

Rodin. The first refinement step led to a total of 14 proof obligations that had to be discharged. While five of them could be proven fully automatically and some further proofs needed only minor effort, the remaining ones required rather complex interactions like manually entered hypotheses or case splitting. Furthermore, the event-B model required additional invariants. This became particularly crucial for the second refinement which, due to the non-atomic nature of the refinement conducted here, could not be modelled in a straight-forward fashion in event-B.

In contrast, both steps could be automatically verified in negligible time by the approach proposed in this chapter. The non-atomic refinement did not lead to an increased run-time in this case.

The second example is a mechanical press controller (MPC), which has also been used to evaluate the automatic verification approach in [20] with the tool ProB. It describes a mechanical press with a motor, a clutch, and a door which interact in such a way as to guarantee a safe use. As in [20], we have modelled the first three refinement steps in UML and verified them with the proposed SMT-based approach. Here, the refinement contains the introduction of new attributes and constraints as well as atomic operation refinement. All three refinement steps have been proven to be correct.

Again, the size of the abstract model and its refinements is shown in Table 1.1 (denoted by MPC0, MPC1, MPC2, and MPC3). In contrast to the first example, the amount of attributes and operations as well as the number of OCL constraints increases. Especially the growing number of operations is important, since, for the SMT-based approach, each of these operations has to be verified according to the criteria presented earlier.

Table 1.2 shows the run-times of our experiments compared to those of ProB. The first two columns indicate which models have been verified against each other. The third and fourth columns contain the run-times of ProB without and with XSB Prolog taken from [20]. In [20], the ProB tool has already been compared to an automatic refinement verification approach based on CSP (namely [18]) which was clearly outperformed by ProB.

Again, the proposed approach proved the correctness of all three refinement steps in negligible time whereas the run-time of ProB was much larger. Also, with and without XSB Prolog, ProB's run-time increased drastically with every step in the

Table 1.2 Experimental results

Abstract	Refined	Run-time		
		ProB (s)	ProB+XSB (s)	SMT-based (s)
AC0	AC1	-	-	< 0.01
AC1	AC2	-	-	< 0.01
MPC0	MPC1	6.28	2.85	< 0.01
MPC1	MPC2	70.57	26.66	< 0.01
MPC2	MPC3	333.85	136.12	< 0.01

refinement process. A similar development has not been observed for the SMT-based method so far.

These experiments confirm that our approach is robust in such a way that it is applicable to various types of models and refinements. Neither errors in the refinement process nor the type of operation refinement—atomic or non-atomic—have a significant influence on the run-time.

1.7 Discussion: Extraction of a Refinement Relation

While the approach presented in Sect. 1.5 serves very well to prove the consistency of a given refinement relation, it may not always be applicable. In order to verify a refinement step, a refinement relation is necessary; otherwise, none of the verification objectives can be checked. However, such a relation may not be present in case that several designers are involved in the modelling process or the refinement process has not been documented.

In this case, methods to extract a refinement relation from the given models are required. The goal of such an extraction is not to obtain any arbitrary relation, but a correct relation based on the criteria presented in Sect. 1.5.1. In the following, we will discuss some related approaches from the literature before sketching some ideas how such an extraction could be realized in the setting of this chapter.

1.7.1 Existing Approaches

In the past, different approaches to retrieve traceability or refinement information have been proposed. Several works focus on information retrieval techniques [4, 19, 22]. Here, the basic idea is to identify textual similarities which may refer to the same concepts. Some of these works focus on relations between different levels of abstraction, e.g. between code and documentation. However, since information retrieval relies on textual similarities, re-naming model elements is a huge problem which might well occur during refinement. Egyed presents a structural analysis

to determine traceability links in [15]. He uses abstraction rules to map classes, attributes, and association. This method works on UML only without considering OCL constraints specifying the operations' behaviour. Briand et al. discuss the use of information gathered by monitoring the designer's modifications as a means to retrieve traceability links in [8]. Like in the approaches mentioned so far, the model's behaviour is not considered in particular.

The authors of [14] propose an adaptation of the algorithm from [26] by Robinson. They apply Robinson's approach to Z refinements and encode it in a model checker. Another extension of the same algorithm can be found in [27], relying on the same mechanism. A relation R containing all potential mappings is step-wise reduced by incorrect mappings until either a correct relation is determined or the all mappings have been removed. Although these approaches do in fact consider the specified behaviour, depending on the variation of the algorithm, the whole system has to be simulated. Since in the beginning R contains all pairs of states, the method does not scale to larger systems.

1.7.2 SMT-Based Relation Extraction

The related approaches discussed above are either of heuristic nature—and therefore incomplete—or they try to solve the problem in an exact way. In the latter case, representing the refinement relation explicitly is infeasible for larger models. Since the encoding of the refinement verification in SMT has proven very successful in terms of scalability, the question is if this approach could also be used to extract a correct refinement relation.

To understand the complexity of the problem, it is useful to view it in the context of automatic synthesis. Verifying a model wrt. some specification is conceptually easier than synthesizing a model which satisfies this specification. In our case, this is reflected in the complexity of the SMT encoding needed to solve the respective problem. The verification of a given refinement relation can be encoded in a purely existentially quantified formula, that checks or falsifies the existence of some pairs of states which violate the refinement relation. Intuitively, the extraction of a correct relation demands one quantifier more: Does there exist some relation such that for all pairs of states it verifies the refinement of our models? Thus, the problem cannot be solved in a complete manner using a quantifier-free SMT encoding.

While there are some solvers that support quantified formulae—such as Z3 [13]—the run-time and memory foot-print increase significantly with each additional quantifier alternation. Alternatively, a two-stage approach can be used that relies solely on quantifier-free encodings:

1. Find some pair of states and a relation which proves their refinement
2. Check if the found relation is a correct refinement relation
3. If yes, we are done. Otherwise continue with step 1

In the sketched algorithm, possible relations are enumerated by the underlying solver until a correct refinement is found. The verification in step 2 has already been solved in this chapter. The algorithm terminates in case of success or if no more relation can be found in step 1. In the latter case, we can be sure that the two models do not represent a correct refinement.

The effectiveness of the sketched approach critically depends on the first step. In the worst case, the algorithm will enumerate a huge number of incorrect relations that will be rejected by the second step. Additional constraints can help to reduce the number of iterations, but will in the same time increase the complexity of the first step. As a promising direction, *sequence diagrams*—representing test cases of the refined model—can be used to narrow down the set of candidate relations. If a correct refinement exists, it must also be applicable on a feasible run of the two models. This approach is subject to ongoing and future research.

1.8 Conclusions

In this chapter, we proposed an automatic approach which proves refinements of UML or SysML class diagrams. By this, we are considering the typical model-driven design flow which usually assumes an initial (abstract) model that is iteratively refined to a more precise representation. Based on a theoretical foundation, we introduced an SMT encoding checking whether the respective refinement relation represents a dbs-simulation and, hence, preserves (safety) properties from the abstract model to the refined model. We compared our approach to the tool ProB, which performs automatic refinement verification on B models. An experimental evaluation has shown that the SMT-based technique can verify refinements much faster and scales better than the B-based method.

For future work, we plan to extend our approach in order to support more modelling elements such as refinement of associations or parameters.

Acknowledgements This work was supported by the Graduate School SyDe (funded by the German Excellence Initiative within the University of Bremen's institutional strategy), the German Federal Ministry of Education and Research (BMBF) within the project SPECifIC under grant no. 01IW13001, as well as the German Research Foundation (DFG) within the Reinhart Koselleck project under grant no. DR 287/23-1 and a research project under grant no. WI 3401/5-1.

References

1. Abrial, J.R.: The B-book: Assigning Programs to Meanings. Cambridge University Press, New York (1996)
2. Abrial, J.R.: Modeling in Event-B: System and Software Engineering, 1st edn. Cambridge University Press, New York (2010)

3. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: UML2Alloy: A challenging model transformation. In: International Conference on Model Driven Engineering Languages and Systems, pp. 436–450. Springer, New York (2007)
4. Antoniol, G., Canfora, G., Casazza, G., Lucia, A.D., Merlo, E.: Recovering traceability links between code and documentation. IEEE Trans. Softw. Eng. **28**, 970–983 (2002)
5. Ben Ammar, B., Bhiri, M.T., Souquières, J.: Incremental development of UML specifications using operation refinements. Innov. Syst. Softw. Eng. **4**(3), 259–266 (2008). doi:10.1007/s11334-008-0056-1
6. Boiten, E.A.: Introducing extra operations in refinement. In: Formal Aspects of Computing, Springer London, pp. 1–13. Springer, London (2012)
7. Braunstein, C., Encrenaz, E.: CTL-property transformations along an incremental design process. Int. J. Softw. Tools Technol. Transfer **9**(1), 77–88 (2006). doi:10.1007/s10009-006-0007-9
8. Briand, L.C., Labiche, Y., Yue, T.: Automated traceability analysis for uml model refinements. Inf. Softw. Technol. **51**, 512–527 (2009)
9. Brummayer, R., Biere, A.: Boolector: an efficient SMT solver for bit-vectors and arrays. In: Tools and Algorithms for Construction and Analysis of Systems, pp. 174–177. Springer, Berlin (2009)
10. Bulychev, P., Konnov, I.V., Zakharov, V.A.: Computing (bi)simulation relations preserving CTL_X^* for ordinary and fair kripke structures. In: Mathematical Methods and Algorithms, vol. 12, pp. 59–76. Institute for System Programming, Russian Academy of Science (2006)
11. Cabot, J., Clarisó, R., Riera, D.: Verification of UML/OCL class diagrams using constraint programming. In: IEEE International Conference on Software Testing Verification and Validation Workshop, pp. 73–80 (2008)
12. Cadoli, M., Calvanese, D., Giacomo, G.D., Mancini, T.: Finite Model Reasoning on UML Class Diagrams Via Constraint Programming. In: R. Basili, M.T. Pazienza (eds.) AI*IA. Lecture Notes in Computer Science, vol. 4733, pp. 36–47. Springer, Berlin (2007)
13. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08/ETAPS'08, pp. 337–340. Springer, Berlin/Heidelberg (2008). URL http://dl.acm.org/citation.cfm?id=1792734.1792766
14. Derrick, J., Smith, G.: Using model checking to automatically find retrieve relations. Electron. Notes Theor. Comput. Sci. **201**, 155–175 (2008)
15. Egyed, A.: Consistent adaptation and evolution of class diagrams during refinement. In: Fundamental Approaches to Software Engineering (2004)
16. Glabbeek, R.: The linear time - branching time spectrum. In: J. Baeten, J. Klop (eds.) CONCUR '90 Theories of Concurrency: Unification and Extension. Lecture Notes in Computer Science, vol. 458, pp. 278–297. Springer, Berlin/Heidelberg (1990)
17. Gogolla, M., Kuhlmann, M., Hamann, L.: Consistency, independence and consequences in UML and OCL models. In: Tests and Proofs, pp. 90–104. Springer, Berlin (2009)
18. Goldsmith, M., Roscoe, B., Armstrong, P.: Failures-Divergence Refinement - FDR2 User Manual (2005)
19. Hayes, J.H., Dekhtyar, A., Osborne, J.: Improving requirements tracing via information retrieval. In: IEEE International Requirements Engineering Conference (2003)
20. Leuschel, M., Butler, M.: Automatic Refinement Checking for B. In: International Conference on Formal Engineering Methods (2005)
21. Loiseaux, C., Graf, S., Sifakis, J., Bouajjani, A., Bensalem, S., Probst, D.: Property preserving abstractions for the verification of concurrent systems. Form. Method. Syst. Des. **6**(1), 11–44 (1995)
22. Natt och Dag, J., Regnell, B., Carlshamre, P., Andersson, M., Karlsson, J.: A feasibility study of automated natural language requirements analysis in market-driven development. Requir. Eng. **7**, 20–33 (2002)
23. Nejati, S., Gurfinkel, A., Chechik, M.: Stuttering abstraction for model checking. In: Software Engineering and Formal Methods, pp. 311–320. Springer, Berlin (2005)

24. Pons, C., Garcia, D.: Practical verification strategy for refinement conditions in UML models. In: Advanced Software Engineering: Expanding the Frontiers of Software Technology. IFIP International Federation for Information Processing, vol. 219, pp. 47–61. Springer, Berlin (2006)
25. Ranzato, F., Tapparo, F.: Computing stuttering simulations. In: Concurrency Theory (CONCUR). Lecture Notes in Computer Science, vol. 5710, pp. 542–556. Springer, Berlin (2009)
26. Robinson, N.J.: Finding abstraction relations for data refinement. Technical Report, Software Verification Research Center, The University of Queensland (2003)
27. Robinson, N.J.: Incremental derivation of abstraction relations for data refinement. In: Formal Methods and Software Engineering. IEEE Computer Society, Los Alamitos (2003)
28. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language reference manual. Addison-Wesley Longman, Essex (1999)
29. Snook, C., Butler, M.: UML-B: formal modeling and design aided by UML. ACM Trans. Softw. Eng. Methodol. **15**(1), 92–122 (2006)
30. Soeken, M., Wille, R., Kuhlmann, M., Gogolla, M., Drechsler, R.: Verifying UML/OCL models using Boolean satisfiability. In: Design, Automation and Test in Europe, pp. 1341–1344. IEEE Computer Society, New York (2010)
31. Soeken, M., Wille, R., Drechsler, R.: Verifying dynamic aspects of UML models. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1–6 (2011)
32. Warmer, J., Kleppe, A.: The Object Constraint Language: Precise Modeling with UML. Addison-Wesley Longman, Boston, MA (1999)
33. Weilkiens, T.: Systems Engineering with SysML/UML: Modeling, Analysis, Design. Morgan Kaufmann, San Francisco, CA (2008)
34. Woodcock, J., Davies, J.: Using Z: Specification, Refinement, and Proof. Prentice-Hall, Upper Saddle River, NJ (1996)

Chapter 2
Towards Simulation Based Evaluation of Safety Goal Violations in Automotive Systems

Oezlem Karaca, Jerome Kirscher, Linus Maurer, and Georg Pelz

Abstract With the advent of the ISO 26262 it became crucial to prove that electrical and electronic products delivered into safety-related automotive applications are adequately safe. For this purpose safety goal violations due to random hardware faults need to be evaluated. In order to gain evident results for argumentation within the evaluation, a fault injection based approach is utilised. Potential risk scenarios are initiated by injection of analogue and digital faults into the heterogeneous behavioural model which comprises the safety-related hardware. For fault injection in heterogeneous models, we propose analogue saboteurs, designed in VHDL-AMS, by which amongst electrical or mechanical, diverse energy domain analogue hardware faults may be injected. For demonstration of this approach, a hardware model, comprising lithium-ion battery cells with a cell balancing and monitoring module and safety-related circuitry is used.

2.1 Introduction

The functional safety standard for road vehicles ISO 26262 addresses the possible hazards caused by failures of electrical and/or electronic (e/e) safety-related systems. In the concept phase (part 3) of the safety life-cycle (part 3–7) [23, 26], possible hazards are identified and subsequently safety goals and safety requirements are formulated. Thereby, safety goals (SGs) are top-level negative requirements assigned to the target hardware element (e.g. battery cells). One of the objectives of the subsequent system design phase (part 4–6) is to verify the effectiveness of the implemented safety mechanisms (measures for avoidance of random hardware failures during operation, e.g. watchdog, redundancy implementation by secondary sensor path) and whether the hardware design complies with

O. Karaca (✉) • J. Kirscher • G. Pelz
Infineon Technologies AG, Neubiberg, Germany
e-mail: oezlem.karaca@infineon.com; jerome.kirscher@infineon.com

L. Maurer
Bundeswehr University Munich, Neubiberg, Germany

© Springer International Publishing Switzerland 2016

F. Oppenheimer, J.L. Medina Pasaje (eds.), *Languages, Design Methods, and Tools for Electronic System Design*, Lecture Notes in Electrical Engineering 361,
DOI 10.1007/978-3-319-24457-0_2

the safety requirements. The standard focuses within the clause 5–9 on evaluation of safety goal violations due to random hardware element faults, in order to evaluate the risk of possible hazards caused by these failures [23]. Recently, there have been a number of investigations on safety analysis in formal model-based manner [2, 10, 11, 15, 21]. Analogue systems with additional fault models cause increasing complexity of the formal model. Therefore, the formal model-based method is only suited for safety analysis of analogue system models that are not highly complex, e.g. in the early phases of the design. To address this issue, there are several proposals for analogue fault injection and simulation. Analogue fault modelling and injection depends mostly on the abstraction level of the nominal model and means a trade-off between simulation results' accuracy and simulation speed, which has been an unchanged challenge since early investigations [25]. In the past years, numerous investigations have been conducted on modelling and injection of hard faults (opens, shorts), soft faults [single event transients (SET)] and fabrication induced faults (statistical parametric variations). The most common techniques proposed to model these faults are by parametric implementation (faults caused by fabrication induced dispersions), by mutations, by saboteurs and their combinations [1, 4, 12–14, 17, 20, 28, 29]. Fault injection by mutations needs initial manual manipulation of the nominal model of a component, however saboteurs may act at the signal interfaces of the hardware (HW) component. In opposite to the discrete digital failure modes (e.g. stuck-at, single event upset, bit-flip), in the analogue domain so far no broadly exercised fault models were established which is due to a greater variety of fault manifestations as a failure mode (continuous signal values) and greater simulation expenses [18]. Therefore different approaches have been proposed in order to enhance efficiency of fault simulations. Earlier, investigations such as [6, 18] proposed inductive fault analysis for analogue faults to reduce the fault list for simulation, where faults which are unlikely to occur are eliminated from the fault list. Other hierarchical approaches focus on fault sensitivities [24] and clustering of critical faults in order to eliminate redundant fault simulations and to reduce simulation time by injecting clustered faults into models of higher levels of abstraction [9, 19, 27]. Recently, behavioural modelling is proposed in fault simulation approaches due to their significantly lower simulation expenses [1, 17, 20, 28, 29].

This work is proposed to contribute to safety analysis in terms of quantification of cause–consequence relationships between random hardware element failures and safety goal violations within specified items. In order to quantify this cause–consequence relationship, faults are injected to the nominal, i.e. fault-free, behavioural model comprising of the equipment under control (EUC) and the safety-related element of interest and subsequent fault simulation and evaluation.

As shown in Fig. 2.1, the item model may comprise amongst elements which can be modelled by digital signals, elements which need to be modelled by analogue signals, e.g. electrical, thermal, mechanical, etc. In this work, this is addressed by heterogeneous behavioural models and injection of multiple energy domain analogue faults. The fault models are in principle of the type saboteurs, by which hard and soft hardware faults may be injected.

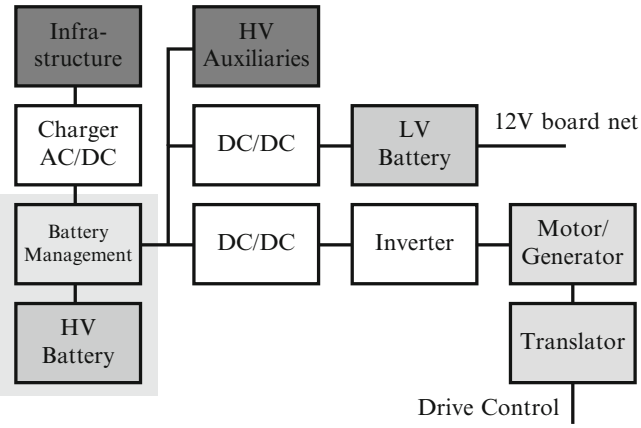


Fig. 2.1 Hardware elements of an electrical drive train

In later stages of the hardware development phase, models of, for instance, analogue, and mixed-signal integrated circuits may already represent very complex systems and fault injection in such systems may be an intricate task, not only due to rising fault simulation expenses but additionally, due to manual fault modelling and injection expenses. In this work this is addressed by

1. fault injection to the analog-mixed-signal VHDL-AMS behavioural model of the hardware design,
2. injection of component faults derived from a quantitative inductive failure analysis, in order to reduce redundant fault simulations,
3. generic fault models, where each instance of a fault model can adapt the desired fault mode by generic configuration, and
4. automatic fault injection by an executable design automation script for fault implementation in the Cadence Virtuoso design environment and subsequent fault activation, i.e. fault injection, in a simulation automation platform [22].

In the second section of this work, the motivation for this approach is clarified. The third section states the proposed fault injection technique. In the fourth section, a case study is presented where the previously described fault injection technique is applied to. The case study comprises a heterogeneous model of an automotive battery management system (BMS) module including twelve lithium-ion cells. Finally, in the last section we sum up the results and experience from the case study and give an outlook for future work.

2.2 Motivation

At the beginning of the safety life-cycle, items are defined which implement certain functions at the vehicle level (e.g. electric drive train) (clause 3–5). An item is a set of HW elements, relating at least a sensor, controller and actuator with each other.

The item addressed in this work is the electric drive train, see Fig. 2.1. In particular the BMS and the battery cells are considered. In this context, respective to the BMS, safety goals are assigned to the HV battery. In this work, a heterogeneous behavioural analogue fault injection method is investigated for the evaluation of safety goal violations due to random hardware element failures. This is used for assessing the effectiveness of implemented safety mechanisms (SMs) when high safety levels are required.

The automotive safety integrity level (ASIL), graded from A to D in order of increasing stringency, is associated with the item and states the safety goal which needs to be achieved by the item, see [23, 26]. Criteria, for the achievement of safety goals, are the single-point and multiple-point fault metrics, as well as the overall residual failure in time (FIT) rate, which need to be evaluated for each safety-relevant item. It is then mapped to an ASIL by comparing the results of the evaluation of the criteria with the target values for the required ASIL for the respective application.

Random hardware element fault classification states the preceding task to mapping an item to ASIL. By the classification, likely hardware element faults are differentiated with respect to their potential to cause hazard [23], assuming that all faults are independent and follow the exponential distribution

$$F(x) = \int_{-\infty}^{x} f_\lambda(t)\mathrm{d}t = 1 - \mathrm{e}^{-\lambda x}, \tag{2.1}$$

for $x \geq 0$ and $F(x) = 0$ for $x < 0$. The rate parameter λ is the failure rate:

- Single-point fault (SPF): potential to directly violate an SG; no SM implemented in the HW element.
- Residual fault (RF): potential to directly violate an SG; at least one SM implemented in the HW element but does not prevent this fault from violating an SG.
- Multi-point fault, Detected (MPF,D): potential to directly violate SG when no SM implemented or to indirectly violate an SG in combination with an independent fault; detected by SM and has no potential to singly violate the SG.
- Multi-point fault, Perceived (MPF,P): potential to directly violate SG when no SM implemented or to indirectly violate an SG in combination with an independent fault; perceived by the driver and has no potential to singly violate the SG.
- Multi-point fault, Latent (MPF,L): potential to directly violate SG when no SM implemented or to indirectly violate an SG in combination with an independent fault; neither detected by SM nor perceived by the driver and has no potential to singly violate the SG.
- Safe fault (SF): no potential to violate directly an SG; in most cases, MPF of order $n > 2$ can be neglected due to their little occurrence probabilities, however faults with very high failure rates or poor diagnostic coverages are exceptions.

The failure rate λ of each safety-related hardware element is then the sum of the failure rates of classified faults [23]

$$\lambda = \lambda_{SPF} + \lambda_{RF} + \lambda_{MPF} + \lambda_{SF}. \tag{2.2}$$

According to the standard, each failure rate and in particular the total failure rates of residual faults and latent faults need to be determined based on estimated values of diagnostic coverage $K_{DC,RF}$ and $K_{DC,MPF,L}$ of safety mechanisms against single-point and multiple-point faults with respect to safety goal violations, see Eqs. (2.3) and (2.4), respectively [23].

$$\lambda_{RF} \leq \lambda_{RF,\text{est}} = \lambda \cdot \left(1 - \frac{K_{DC,RF}}{100}\right) \tag{2.3}$$

$$\lambda_{MPF,L} \leq \lambda_{MPF,L,\text{est}} = \lambda \cdot \left(1 - \frac{K_{DC,MPF,L}}{100}\right). \tag{2.4}$$

Based on this classification of random hardware element faults, hardware architectural metrics for single-point/residual faults and latent multiple-point faults are calculated which are needed for mapping the application to ASIL, see Annex C of part 5 in [23]. The metrics state the robustness of the hardware against random hardware faults with respect to safety mechanisms and design measures in order to prevent these faults. For this reason, these metrics are decisive for the assessment of functional safety of the application. Nevertheless, in order to assess functional safety, methods based on estimations are not sufficient for applications with safety goals aimed to achieve ASIL C or D. Here, methods for the evaluation of diagnostic coverage with respect to the elements' types are proposed, by which faults or failure modes for certain elements are analysed for derivation of diagnostic coverage. Additionally, dealing with analogue and mixed-signal circuit designs and respective models with analogue continuous signals and continuous fault propagation to failure modes in the presence of random hardware element faults, safety goal violations are difficult to evaluate by pure estimation and formal model-based approaches. Even if this is possible in some cases, still profound evidence is needed for argumentation in the safety assessment. In this context, it is necessary to perform analogue and mixed-signal fault simulations in order to gain evidence for evaluation of safety goal violations.

2.3 Fault Injection by Analogue Saboteurs

The generic fault models described in this section are of type saboteurs and are designed to inject analogue hard faults like open-circuit or short-circuit and soft faults like transient faults or offset faults. Within each hardware element, components state certain functions (e.g. A/D conversion) and interact with other

components or elements via interface signals. Each component failure mode is the effect of probable faults within the component, causing deviations of the interface signals from correct operation values (errors). Injection of faults in accordance with component failure modes does require knowledge of the component's internal error propagation in order to derive accurate failure modes and as a consequence accurate fault models to begin with fault injection in the first place. One approach is to apply stand-alone component fault simulations where faults are injected inside the components architecture, subsequently simulated stand-alone in an adequate test-bench and derived again to the component's interface signals. Eventually, this can be done for a reasonable choice of architectural levels within each component. An alternative approach is to utilise component failure modes which were previously determined within an inductive analysis, like failure mode and effect analysis (FMEA), and to inject faults in accordance with these failure modes.

In order to gain evidence for the evaluation of safety goal violations, component failure modes, derived from an FMEA, were injected into the behavioural model used in the case study. However, when dealing with multiple energy domain systems in certain items, the proposed fault modelling technique may also be applied to failure modes of actuating or sensing elements of the item. Although the failure rates of non e/e systems are not included within the standard in terms of hardware architectural metric calculation, the integrity of e/e safety-related systems in the presence of associated, e.g. mechanical failure modes, needs to be analysed and included in the hardware architectural metric calculation. In this section basic considerations for multiple energy domain generic analogue fault model design and implementation in VHDL-AMS are discussed. Additionally, a method for automatic fault injection is topic of investigation which is based on the scripting language SKILL in the Cadence Virtuoso Design environment.

2.3.1 Diverse Energy Domain Saboteurs

A behavioural model representing an item in automotive applications may comprise elements of diverse energy domains, see, for instance, Fig. 2.1. Amongst digital signals, e/e elements can be typically modelled by analogue electrical signals, with respect to the duality of voltage and current. However, particularly when including models of actuator and sensor elements, further analogue signals of other energy domains such as mechanical (translational/rotational), magnetic and thermal are involved. Other energy domain analogue faults can be implemented in accordance with component failure modes into the model in the same fashion as electrical failure modes. For this purpose the elementary analogies of diverse conservative energy systems [5] are utilised, with regard to modelling, for instance, actuating/sensing elements, in accordance with Kirchhoff's circuit laws. In contrast to digital fault modelling techniques, in the analogue domain, a hard or soft fault inside an analogue component will not propagate to a discrete component failure mode (like stuck-at 1). The manner in which the fault will manifest itself in a failure mode is determined by continuous disturbances of the signal attributes amplitude, phase and frequency. This circumstance marks the challenge in analogue fault injection by saboteurs.

In this work, the fault models are generic adjustable and comprise fault modes which are limited to open-circuit, short-circuit and SET/offset. An open-circuit fault can be used, for instance, in the electrical domain to model a stuck-at high or low failure mode of a comparator by injecting the open-circuit fault either to the component's low-input or high-input, respectively. In the mechanical domain the open-circuit fault model may be used, for instance, to inject load transmission failure due to mechanical overload. The open-circuit fault model may also be used, for instance, to model an overheating failure due to a defect of heat transfer paste of a cooling element. Further examples for fault, error and failure relation in different energy domains are shown in Table 2.1, see [16] for extended table (including effort aspect temperature T and flow aspect P in thermal domain as well as effort aspect displacement s or velocity v and flow aspect force F in translational mechanical domain). In this work, we focus on analogue fault injection into the behavioural model by generic fault models of the type saboteur.

Table 2.1 Comparison of typical examples of fault, error and failure relation in electrical and rotational mechanical which may be represented by the fault modes of the generic saboteurs in a model

Energy domains	Electrical	Rotational mechanical
Effort aspects	Voltage u	Angle φ, Angular velocity α
Flow aspects	Current i	Torque M
Fault modes	*Examples of hardware fault, error and failure mode relationship*	
Open-circuit	*Fault:* el. isolated ADC input *Error:* (non-)inv. voltage *Failure:* stuck-at H/L	*Fault:* motor torque shaft break *Error:* no load transfer *Failure:* idle running motor
	Explanation: failure during load transfer from point A to B in the presence of effort aspects $\{u, s/v, \varphi/\alpha, T\}_{A-B} \neq 0$ *resulting in flow aspects* $\{i, F, M, P\}_{A-B}$ *very low to zero*	
Short-circuit	*Fault:* bridged ADC input *Error:* writes wrong output *Failure:* reads const. input 0V	*Fault:* radial load applied to torque shaft *Error:* load distribution *Failure:* overloading of motor
	Explanation: failure during load transfer from point A to B with additional undesired coupling to point C in the presence of effort aspects $\{u, s/v, \varphi/\alpha, T\}_{A/B-C}$ *resulting in flow aspects* $\{i, F, M, P\}_{A/B-C} \neq 0$	
Single event transient (SET), offset	*Fault:* EMI to ADC pin(s) *Error:* writes wrong output *Failure:* oscillation	*Fault:* impact applied to torque shaft *Error:* load distribution *Failure:* load transfer oscillates
	Explanation: failure during load transfer from point A to B where effort aspects $\{u, s/v, \varphi/\alpha, T\}_{A-B}$ *or flow aspects* $\{i, F, M, P\}_{A/B-C}$ *experience a transient disturbance*	

2.3.2 Design of Generic Saboteurs and Injection into Nominal Model

The analogue saboteurs are designed in VHDL-AMS and used in the Cadence Virtuoso Schematic Editor environment. Faults are injected into the schematic design/model by manipulation of the net-list. Additionally, we aim to simplify the fault injection procedure by modelling saboteurs with generic attributes. This means that a generic saboteur located at a certain component pin may optionally adapt different faults, that is fault modes, including fault-free mode.

2.3.2.1 Effect-Based Analogue Saboteur Design

A fault in an analogue circuit may generally be experienced by failure of a signal, with respect to its duality regarding effort and flow aspects. We propose to manipulate effort aspects and flow aspects in order to implement a fault. This can be enabled by designing the fault model based on digitally controlled analogue switches. Figure 2.2 shows the basic structure and respective behaviour in semi-formal notation:

- $flow_{net} = 0.0,\ \ flow_{short} = 0.0;\ \ \text{if}\ \ mode = \text{open}$
- $effort_{net} = f(),\ \ flow_{short} = 0.0;\ \ \text{if}\ \ mode = \text{trans./offset}$
- $effort_{net} = 0.0,\ \ effort_{short} = 0.0;\ \ \text{if}\ \ mode = \text{short}$
- $effort_{net} = 0.0,\ \ flow_{short} = 0.0;\ \ \text{else (nominal)}$

The saboteur's implementation into the desired net-list is performed by interconnecting terminals p_i, p_o to the analogue signal of interest and connecting terminal p_{short} to the desired short-circuit signal.

Digitally controlled analogue switches may also be modelled by simply implementing equations based on

$$\text{effort} = \text{damping} \cdot \text{flow},$$

where the *damping* value is either very high or very low, depending on the fault mode. In spite of the need to determine these values, during fault simulation, exceptions can occur. Depending on the abstraction of the components which are

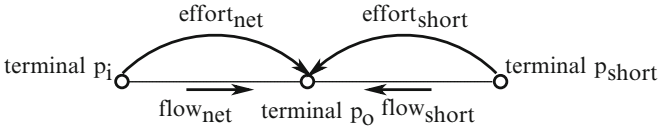


Fig. 2.2 Basic structure of generic saboteur involving effort and flow aspects for diverse energy domains

connected to the fault models, the simulation with open-circuit fault will perform as expected, instead pi and po are always shorted, causing $effort_{net}$ to be zero. This happens in cases where component signals have no flow aspects defined but only the effort aspects, e.g. when an A/D-converter model measures analogue voltage signal, the respective terminals may be sufficiently modelled without flow aspect. With the implementation method used in this work we aim to address this issue in fault injection into abstract models.

2.3.2.2 Implementation in VHDL-AMS

Transferring the fault mode configuration directly to VHDL-AMS code will most likely cause convergence issues during simulation time. This is due to discontinuities in the effort and flow aspects arising from the assumption of ideal digitally controlled analogue switches. Convergence issues are avoided by modifying the saboteur's structure by applying measures in order to deal with the discontinuities arising from the switching states [3]. Other energy domain generic saboteurs are modelled in the very same fashion by defining terminals, effort and flow aspects respective to their nature.

The generic saboteur is controlled by its real-valued generics. First parameter determines the fault mode. We approximate transient faults by using the ramp function as proposed in [17]. Timing of any fault mode is also controlled by parameters, setting the fault injection time and duration. To inject non-transient faults, the parameter, setting the end of injection duration, has to be set to a value beyond simulation time. Terminal pgnd is used to measure the voltage at pi referenced to ground in order to inject an open-circuit fault by setting potential at terminal pi and po equal. Nominal behaviour is implemented by shorting latter two terminals.

2.3.2.3 Fault Injection into Nominal Model and Automation

Depending on the complexity of the item model, it can be troublesome to inject faults manually. We aim to overcome this by fault injection automation and in particular to explore the feasibility of fault injection automation with the generic saboteurs proposed in this work. For this intention, the modularity of the saboteurs is a convenient feature that is used in the automation procedure, due to the fact that they are implemented basically by interconnecting them with the respective signals/nets of respective component instances.

For automatic implementation of saboteurs into the item model, a design automation script in SKILL language is written which is executable within the Cadence Virtuoso environment. The design automation procedure requires three parameters, defining directory paths of the initial test-bench (mixed HDL and spice), of the fault model library and injection directive are passed to the main procedure. First the initial test-bench, comprising the selected components, is duplicated. Then the

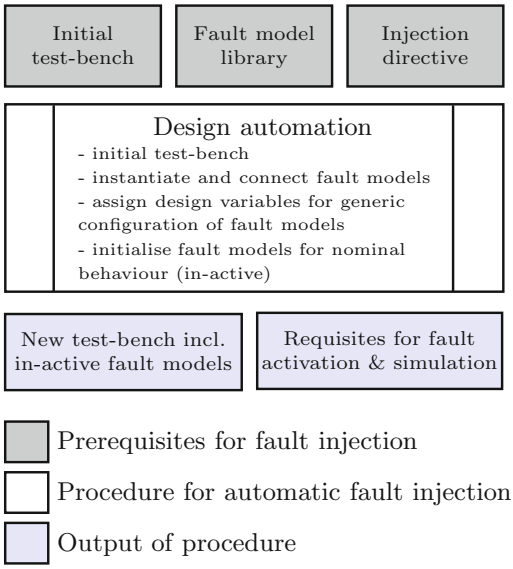


Fig. 2.3 Concept of fault injection automation

test-bench duplicate is used for the fault injection procedure. When all fault models are implemented to the net-list, the test-bench comprises the nominal behavioural model which is extended by generic saboteurs. Per default, no fault is activated, i.e. all faults are in-active. This test-bench with in-active faults is replicated and faults are activated or deactivated for each single fault and multiple-point fault simulation. Figure 2.3 illustrates the concept of the fault injection procedure.

2.4 Case Study: Automotive BMS and EUC

The previously described fault injection technique is applied to a schematic test-bench which comprises the behavioural model of a lithium-ion (Li-ion) battery management integrated circuit (IC) module, including Li-ion cell models, see Fig. 2.4. The circuit monitors temperatures and voltages for twelve Li-ion cells and balances charge across the cells using active balancing or passive balancing [7, 8]. Additionally, constant current battery charging can be simulated. The main analogue and mixed-signal functions of the IC are covered by 12 individual parallel 13-bit Δ-Σ converters for primary voltage measurement and a 10-bit successive approximation register (SAR) converter for secondary voltage measurement. The primary voltage measurement is used for precise voltage reading for cell balancing and the secondary voltage measurement is used for fast cell over-voltage and under-voltage detection. In this context the primary converter is the safety-related hardware element and the secondary converter is its safety mechanism, dedicated to the prevention of safety goal violations due to over-voltage or under-voltage and acts in case of a failure of the primary converter, in order to prevent subsequent potential hazard.

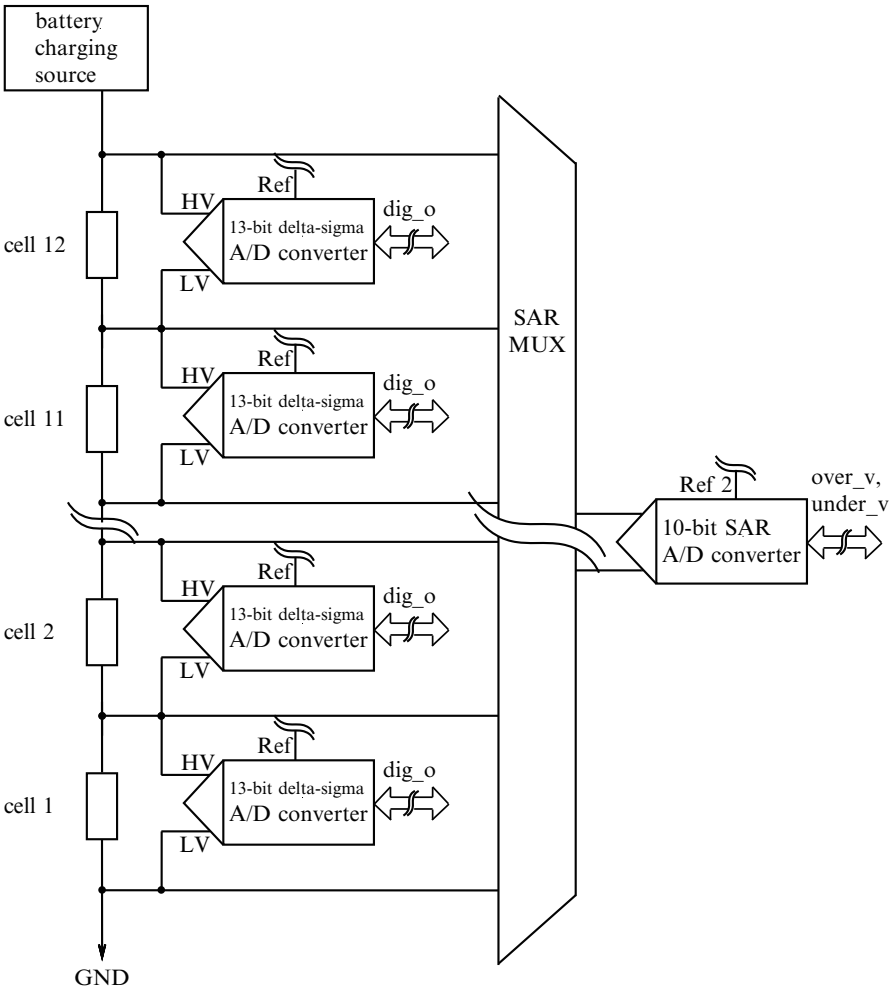


Fig. 2.4 Simplified extract of the nominal schematic test-bench of the behavioural model of the battery management element, including battery cell models

From the safety analysis perspective it is of interest to monitor the Li-ion battery cells, in the presence of hardware faults in the IC. In the case study, the cell voltages are monitored and faults are injected by generic saboteurs. Some stringent safety goals for the Li-ion cells whose potential for being violated can be evaluated by simulation are

- SG1: avoid over-discharge of any battery cell,
- SG2: avoid overcharge of any battery cell,
- SG3: avoid cell short-circuit.

By simulation, evidence for violation can be evaluated by monitoring the cell voltages and currents [26]. If the monitored values exceed or fall below the defined safety requirements, respectively, a potentially hazardous random hardware element fault is identified.

2.4.1 Test-Bench Set-Up

The test-bench is set up for constant current charging operation, with 12 parametrised Li-ion cells with different state of charges (SOCs). Charging high energy Li-ion cells takes hours in real life. However, in the simulation, by scaling the cell capacity to a lower value and adjusting other test-bench parameters respectively, simulation time is reduced without influence on the results accuracy.

In the case study the cells are charged with a 10A constant current and varying SOC initialisation. The charging stops when the Δ-Σ converter reads the end-of-charge voltage value at any cell. In parallel the SAR converter checks the cell voltages and disconnects the cells from the charging source, in case of a malfunctioning Δ-Σ converter, by disabling the safety relay which is in series connected to the battery. The test-bench switches to idle state after the charging process.

2.4.2 Selected Simulation Results for SPF and Discussion

The fault simulation covers 88 single-point analogue and digital faults at selected electronic/electrical components of the safety-related battery management IC respective to the 10th, 11th and 12th cell. Furthermore, SPF, which do not violate any SG are analysed by dual-point fault injection. Each simulation was set to 10s simulation time. The total time required for each transient analysis depended on the injected fault and the simulations took between 10s and 60s. The nominal simulation with in-active faults did not mentionable slow down the transient analysis.

Table 2.2 provides information regarding the test-bench set-up, the fault injection reference, taken from an FMEA source and the proposed fault injection for realising the component failure modes. Due to the fact that the battery management IC comprises a safety mechanism stated by the SAR converter, all SPFs which lead to a safety goal violation can be classified RFs. Specific to this case-study, the residual fault amount can be quantified by the sum of the failure rates of all faults causing an SG violation. Faults which lead to a failure (i.e. any deviation of the component's performance from nominal) but no SG violation are considered for further MPF analysis because of a potential perceived fault (MPF,P) disclosure, in combination with another independent fault. Faults which lead to no failure and hence to no SG violation (not perceived, nor detected) need further MPF analysis because of a potential latent fault (MPF,L) disclosure, in combination with another independent fault.

Table 2.2 Test-bench set-up, components considered for fault injection based on FMEA reference

Test-bench set-up	
SGs respective to	12 Li-ion battery cells; monitoring of all cell's voltage
Safety-related HW	Li-ion battery management IC module
Operating mode	10A constant current charging
Operating cond.	Nominal ambient conditions; SOCs with variable initialisation

Fault injection reference (from FMEA source)		Fault injection
Component	Component failure mode⟶ effect description	Fault mode
Δ-Σ converter (one component for each cell)	Stuck-at high⟶ only non inverted differential signal	el. open-circuit
	Stuck-at low⟶ only inverted differential signal	el. open-circuit
	Drift (high/low)⟶ wrong diff. voltage	el. across disturb.
	DC-fault⟶ short between HV and LV part	el. interconn.
	Transient spike p/m⟶ differential voltage oscillating	el. across disturb.
SAR converter (one component for each cell)	Output stuck-at 1⟶ stuck-at max. output value	dig. stuck-at 1
	Output stuck-at 0⟶ stuck-at min. output value	dig. stuck-at 0
	Switching point wrong⟶ 10 % deviation	el. across disturb.
	Stuck-at high⟶ only non inverted differential signal	el. open-circuit
	Stuck-at low⟶ only inverted differential signal	el. open-circuit
	Drift (high/low)⟶ wrong diff. voltage	dig. bit-flip
	DC-fault⟶ short between HV and LV part	el. interconn.
	Transient spike p/m⟶ differential voltage oscillating	el. across disturb.
5 V supply regulator	Input to output short⟶ chip over-voltage/destruction	el. interconn.
	Input to output open⟶ no supply for chip	el. open-circuit
	Output too high/low⟶ supply over/under-voltage	el. across disturb.
	Output oscillates⟶ modulated supply voltage	el. across disturb.
Reference voltage (one for all Δ-Σ converters)	Input to output short⟶ reference voltage on supply level	el. interconn.
	Input to output open⟶ no supply for Δ-Σ converter	el. open-circuit
	Output too high/low⟶ wrong Δ-Σ output	el. across disturb.
	Output oscillates⟶ wrong Δ-Σ output	el. across disturb.

Figures 2.5, 2.6, and 2.7 show simulation results for selected component failure modes:

- output stuck-at 0 of SAR converter of 11th cell (fault 42)
- output stuck-at 0 of SAR converter of 12th cell (fault 61)
- DC fault (short) of Δ-Σ converter of 12th cell (fault 74)
- fault 42 and output stuck-at 0 of SAR converter of 12th cell (fault 44) (dual-point fault injection)

In Fig. 2.5 fault injection results respective to equal SOC=50 % initialisation of the cells are drawn. Injection of fault 42 is not perceived from the cell voltages behaviour and nor is it detected by any other safety mechanism. Safety mechanism respective to 12th cell deactivates the charging process by a single pulse of digital

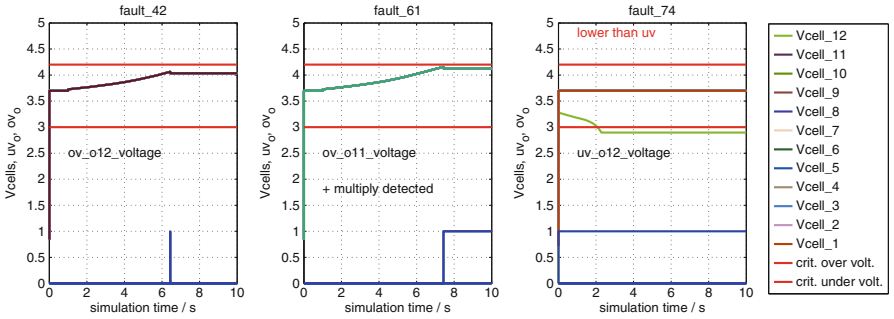


Fig. 2.5 Simulation results for SPF injection. All cells are initialised to equal SOC=50 %. Cell voltages, critical over- and under-voltages are drawn together with the digital over-voltage ov_o and under-voltage uv_o detection. The charging process begins at 1.6*s* and ends when maximum voltage at any cell is reached

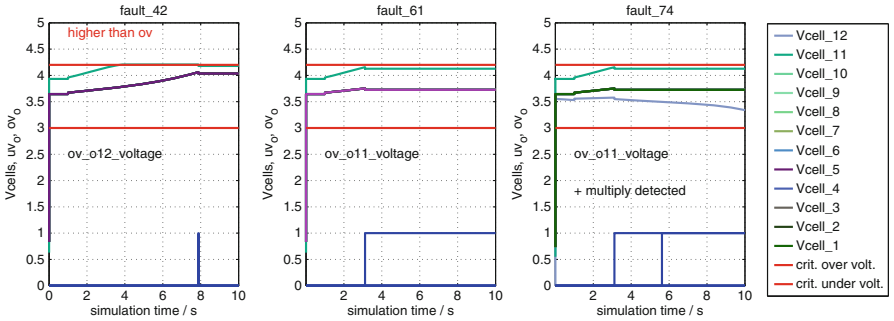


Fig. 2.6 Simulation results for SPF injection. 11th Cell initialised to SOC=80 % and all other cells initialised to SOC=40 %. Cell voltages, critical over- and under-voltages are drawn together with the digital over-voltage ov_o and under-voltage uv_o detection. The charging process begins at 1.6s and ends when maximum voltage at any cell is reached

ov_o when the maximum voltage at 12th cell is reached. In this context, fault 42 is latent and can be classified an SF when we can exclude its potential for contribution to an MPF,L with another independent fault. Injection of fault 61 is perceived and (multiply) detected by another safety mechanism which writes a constant positive over-voltage ov_o signal. The safety mechanism enforces the charging process to stop and prevents the fault from violating SG2. In this context, fault 61 is perceived and detected and can be classified an SF when we can exclude its potential for contribution to an MPF,P or MPF,D with another independent fault. Injection of fault 74 is perceived and detected by the safety mechanism which writes a constant positive under-voltage uv_o signal. Nevertheless, fault 74 violates SG1 and must be classified an RF.

In Fig. 2.6 fault injection results respective to 11th cell SOC=80 % and all other cells SOC=40 % initialisation are drawn. Same as in Fig. 2.5, injection of fault 42 is not detected by any other safety mechanism. However, contrary to Fig. 2.5,

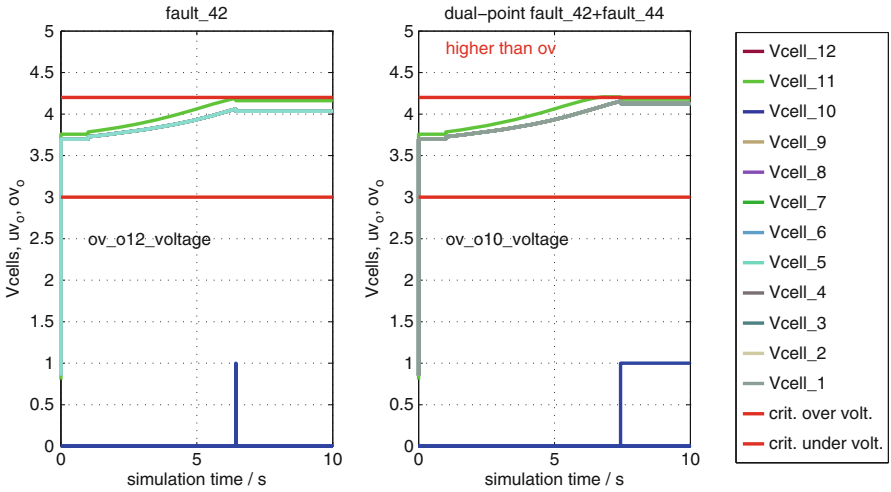


Fig. 2.7 Comparison of single-point and dual-point fault injection. 11th Cell initialised to SOC=60 % and all other cells initialised to SOC=50 %. Cell voltages, critical over- and under-voltages are drawn together with the digital over-voltage ov_o and under-voltage uv_o detection. The charging process begins at 1.6 s and ends when maximum voltage at any cell is reached

fault 42 does violate SG2 and hence must be classified an RF. Same as in Fig. 2.5, injection of fault 61 is perceived and detected by another safety mechanism. Therefore, further analysis to exclude MPF,P or MPF,D potential is required. Same as in Fig. 2.5, injection of fault 74 is detected by the safety mechanism. However, contrary to Fig. 2.5, fault 74 does not violate any SG. Similar results are perceived for all three faults when the cell SOCs differ significantly among themselves.

2.4.3 *Selected Simulation Results for Dual-Point Faults and Discussion*

Latency of fault 42, as shown in Figs. 2.5 and 2.7, requires further analysis of its MPF,L potential, together with another independent fault. For this purpose dual-point fault injection is performed. Figure 2.7 shows that fault 42 does violate together with fault 44 SG2. In this context we can derive a differentiated classification of fault 42:

- RF for cell SOCs which significantly differ among themselves (by a ratio $x : 1$ with $x \geq 2$)
- MPF,L for similar cell SOCs and in combination with fault 44.

From the single-point and dual-point fault simulation results it can be concluded that the effect of random hardware faults can depend on other independent factors

outside the electronics, like the SOCs of the battery cells. As presented in this case-study, this requires a more differentiated evaluation procedure which includes such factors.

2.5 Conclusion and Outlook

To comply with the safety requirements in automotive applications in accordance with the ISO 26262, and in particular when achievement of ASIL C or D is aimed, a method for evident argumentation within the evaluation of safety goal violations due to random hardware failures in electrical and electronic systems is needed. In this work, a simulation-based method is presented, in which automated diverse conservative energy domain analogue and digital fault injection to a heterogeneous behavioural model is applied. The simulation results are used within the random hardware fault classification procedure, which can be used for subsequent quantification of the diagnostic coverage and hence evaluation of the effectiveness of the safety mechanisms.

The presented work comprises a constitutive approach and can be used for further investigations of simulation-based single-point/residual and multiple-point analogue fault injection in order to support evaluation of safety goal violations due to random hardware faults, in accordance with the ISO 26262. Furthermore, it allows a differentiated argumentation in the context of random hardware fault classification, by accounting for additional factors which contribute to the failure severity. Further investigation is needed in order to support plausibility analyses for multiple-point faults and elaboration of diagnostic coverage of safety mechanisms. In the simulations stated in this work, nominal operation conditions, e.g. ambient temperature, were assumed. Additionally, the test-bench was configured for the charging process. However, operation conditions and operation modes may contribute to the exposure and severity of the safety goal violations due to random hardware faults and need therefore further investigation.

Acknowledgements This research project *SafeBatt* is supported by the German Government, Federal Ministry of Education and Research under the grant number 03X4631A.

References

1. Ahmadian, S.M., Miremadi, S.G.: Fault injection in mixed-signal environment Using behavioral fault modeling in Verilog-A. In: IEEE International Behavioral Modeling and Simulation Convergence, BMAS, pp. 69–74 (2010)
2. Aljazzar, H., Fischer, M., Grunske, L., Kuntz, M., Leitner, F., Leue, S.: Safety analysis of an airbag system using probabilistic FMEA and probabilistic counter examples. In: 6th International Conference on the Quantitative Evaluation of Systems, QEST, pp. 299–308 (2009)
3. Ashenden, P.J., Peterson, G.D., Teegarden, D.A.: The System Designer's Guide to VHDL-AMS - Analog, Mixed-Signal, and Mixed-Technology Modeling. Morgan Kaufmann, San Francisco (2003)

4. Bounceur, A., Mir, S., Rolindez, L., Simeu, E.: CAT platform for analog and mixed-signal test evaluation and optimization. In: IFIP International Conference on Very Large Scale Integration, pp. 320–325 (2006)
5. Christen, E., Bakalar, K.: VHDL-AMS - A hardware description language for analog and mixed-signal applications. IEEE Trans. Circuits Syst. II, Analog Digit. Signal Process. **46**(10), 1263–1272 (October 1999)
6. Corsi, F., Morandi, C.: Inductive fault analysis revisited. In: IEE Proceedings-G on Circuits, Devices and Systems, pp. 253–263 (1991)
7. Daowd, M., Omar, N., van den Bossche, P., van Mierlo, J.: Passive and active battery balancing comparison based on MATLAB simulation. In: Proceedings of the IEEE Vehicle Power and Propulsion Conference (VPPC), pp. 1–7 (2011)
8. Einhorn, M., Roessler, W., Fleig, J.: Improved performance of serially connected Li-Ion batteries with active cell balancing in electric vehicles. IEEE Trans. Veh. Technol. **60**, 2448–2457 (2011)
9. Farooq, M.U., Xia, L., Azmadi, F.: A critical survey on automated model generation techniques for high level modeling and high level fault modeling. In: National Postgraduate Conference, NPC, pp. 1–4 (2011)
10. Gudemann, M., Ortmeier, F.: Probabilistic model-based safety analysis. In: 8th Workshop on Quantitative Aspects of Programming Languages, EPTCS, pp. 114–128 (2010)
11. Grunske, L., Colvin, R., Winter, K.: Probabilistic model-checking support for FMEA. In: 4th International Conference on the Quantitative Evaluation of Systems, QEST, pp. 119–128 (2007)
12. Harvey, R.J.A., Richardson, A.M.D., Bruls, E.M.J.G., Baker, K.: Analog fault simulation based on layout dependent fault models. In: Proceedings in International Test Conference, pp. 641–649 (1994)
13. Hopsch, F.: Variation-aware fault modeling. 19th IEEE Asian Test Symposium, ATS, pp. 87–93 (2010)
14. Joonsung, P., Madhavapeddiz, S., Paglieri, A., Barrz, C., Abraham, J.A.: Defect-based analog fault coverage analysis using mixed-mode fault simulation. In: 15th IEEE International Mixed-Signals, Sensors, and Systems Test Workshop, IMS3TW, pp. 1–6 (2009)
15. Joshi, A., Heimdahl, M.P.E.: Behavioral fault modeling for model-based safety analysis. In: 10th IEEE High Assurance Systems Engineering Symposium, HASE, pp. 199–208 (2007)
16. Karaca, O., Kirscher, J., Maurer, L., Pelz, G.: Towards simulation based evaluation of safety goal violations in automotive systems. In: 2014 Forum on Specification and Design Languages (FDL). http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7119346&isnumber=7119333
17. Leveugle, R., Ammari, A.: Early SEU fault injection in digital, analog and mixed signal circuits: a global flow. In: Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, DATE, pp. 590–595 (2004)
18. Milne, A., Taylor, D., Talbot, A.D.: Generation of optimised fault lists for simulation of analog circuits and test programs. In: IEEE Proceedings on Circuites Devices Systems, pp. 355–360 (1999)
19. Nagi, N., Abraham, J.A.: Hierarchical fault modeling for analog and mixed-signal circuits. In: 10th IEEE Design, Test and Application: ASICs and Systems-on-a-Chip, VLSI Test Symposium, pp. 96–101 (1992)
20. Perkins, A.J., Zwolinksi, M., Chalk, C.D., Wilkins, B.R.: Fault modelling and simulation using VHDL-AMS. In: 16th Analog Integrated Circuits and Signal Processing, pp. 141–155. Kluwer, Dordrecht (1998)
21. Pintard, L., Fabre, J.-C., Kanoun, K., Leeman, M., Roy, M.: Fault injection in the automotive standard ISO 26262: an initial approach. In: 14th European Workshop on Dependable Computing, EWDC, pp. 126–133 (2013)
22. Pirker-Fruhauf, A., Kunze, M.: A novel methodology to combine and speed-up the verification process of simulation and measurement of integrated circuits. In: IEEE AUTOTESTCON, pp. 259–262 (2008)

23. Road Vehicles - Functional Safety. International Organisation for Standardisation, ISO 26262, 1st edn. (2011)
24. Singh, M., Koren, I.: Fault-sensitivity analysis and reliability enhancement of analog-to-digital converters. IEEE Trans. Very Large Scale Integr. VLSI Syst. **11**(5), 839–852 (October 2003)
25. Spinks, S.J., Bell, I.M.: "Analogue fault simulation," in Mixed Mode Modelling and Simulation, In: IEE Colloquium on, pp. 9/1–9/5. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=383641&isnumber=8687 (1994)
26. Taylor, W., Krithivasan, G., Nelson, J.J.: System safety and ISO 26262 compliance for automotive lithium-ion batteries. In: IEEE Symposium on Product Compliance Engineering, ISPCE, pp. 1–6 (2012)
27. Voorakaranam, R., Chakrabarti, S., Hou, J., Gomes, A., Cherubal, S., Chatterjee, A.: Hierarchical specification-driven analog fault modeling for efficient fault simulation and diagnosis. In: Proceedings on the 1997 International Test Conference, TEST, pp. 903–912 (1997)
28. Wilson, P.R., Kilic, Y., Ross, J.N., Zwolinski, M., Brown, A.D.: Behavioural modelling of operational amplifier faults using VHDL-AMS. In: Proceedings on Design, Automation and Test in Europe Conference and Exhibition (2002)
29. Zwolinski, M., Brown, A.D.: Behavioural modelling of analog faults in VHDL-AMS - a case study. In: Proceedings of the 2004 International Symposium on Circuits and Systems, ISCAS, pp. 632–635 (2004)

Chapter 3
Hybrid Dynamic Data Race Detection in SystemC

Alper Sen and Onder Kalaci

Abstract Data races are one of the most common problems in concurrent programs. As SystemC standard allows nondeterministic scheduling of processes, this leads to many concurrency problems. Data races are the most commonly encountered concurrency problems and they need to be detected for improving SoC design quality. Different executions of the same concurrent program may lead to unexpected results due to race conditions. We develop a hybrid dynamic data race detection algorithm for SystemC/TLM designs that adopts the well-studied dynamic race detection algorithms; lockset and happens-before. We develop a segment-based technique where a segment is defined as a set of consecutive memory accesses by a single thread. Experiments show that our solution has fewer false positives than lockset and fewer false negatives than happens-before algorithms. Our implementation uses dynamic binary instrumentation allowing us to work on designs for which source codes may not be available such as pre-compiled IPs.

3.1 Introduction

SystemC is one of the most commonly used system level design languages. It is a concurrent language that allows the execution of multiple processes. The language standard allows nondeterministic scheduling of a process from the list of processes that are available for execution. Similar to other concurrent and multithreaded languages/libraries such as Java or Pthreads, designs in SystemC suffer from concurrency related errors. These errors include race conditions, deadlocks, and livelocks. In this paper we focus on detecting race conditions.

A *data race* is said to occur when more than one thread accesses a shared resource, at least one of those accesses is a write and there is no appropriate

A. Sen (✉) • O. Kalaci
Department of Computer Engineering, Bogazici University, 34342 Istanbul, Turkey
e-mail: alper.sen@boun.edu.tr; onder.kalaci@boun.edu.tr

© Springer International Publishing Switzerland 2016

F. Oppenheimer, J.L. Medina Pasaje (eds.), *Languages, Design Methods, and Tools for Electronic System Design*, Lecture Notes in Electrical Engineering 361,
DOI 10.1007/978-3-319-24457-0_3

synchronization between this two accesses (such as locks). In a multithreaded program, not all data races result in abnormal program behavior. Sometimes they are present for performance reasons. However, in most cases they lead to unexpected and wrong behaviors. Data races are difficult to detect because they may not happen in consecutive executions of a program even if the same inputs and initial state are given.

We demonstrate a race condition in Fig. 3.1 adapted from [1]. In this example, the *guard* process prevents *pressure* from reaching *PMAX*. However, since SystemC kernel allows nondeterministic choice between *guard* and *increment* processes, it is possible for *increment* thread to be executed when $pressure = PMAX$, hence exceeding the *PMAX* limit. This condition can be detected by finding the data race on the shared variable *pressure*.

Note that the SystemC standard [11] specifies a strict order on the execution of the processes as follows: "The order in which process instances are selected from the set of runnable processes is implementation-defined. However, if a specific version of a specific implementation runs a specific application using a specific input data set, the order of process execution shall not vary from run to run." However, during early design phases the application goes through several changes.

```
1  #define PMAX 7
2  #include <systemc.h>
3
4  SC_MODULE(mod) {
5  public:
6    int pressure;
7
8    void guard() {
9     while(true) {
10    if(pressure==PMAX) pressure=(PMAX-1);
11    cout << "G:" << pressure << endl;
12    wait(1,SC_NS);
13    }
14   }
15
16   void increment() {
17    while(true) {    pressure++;
18    cout << "G:" << pressure << endl;    wait(1,SC_NS);
19    }
20   }
21
22   SC_CTOR(mod) {
23     SC_THREAD(increment);
24     SC_THREAD(guard);
25     pressure = 0;
26   }
27 };
28
29 int sc_main(int argc, char* argv[]) {
30   mod m("mod");
31   sc_start();
32   return 0;
33 }
```

Fig. 3.1 SystemC Data Race Example 1 [1]

For example, when the location of processes for the shown example is changed, the order of process execution can change. Also, if the designer uses another SystemC simulator or uses a parallel SystemC simulator, which is gaining popularity in the literature, then the order of process execution can change. Hence, the designer should investigate this nondeterministic behavior of the application to prevent concurrency problems.

The literature on race detection in concurrent and distributed programs is vast. The approaches can be summarized as static and dynamic. Static approaches [4, 18] can investigate all execution paths of a program but they are limited in terms of their scalability and must be conservative. Most race detection work focuses on dynamic techniques, which use the execution trace of programs rather than the whole program itself therefore can be applied to real-life designs. There are mainly three dynamic approaches for data race detection, lockset based [22], happens-before based [5, 21], and hybrid [20, 24]. Lockset based algorithms are the fastest, they do not generate false negatives (detector does not produce a warning although there is a race in the program) but they suffer from many false positives (detector produces a warning although there is no race in the program), which is not desirable. Happens-before based algorithms are the slowest, they do not generate false positives but they suffer from false negatives. Hybrid algorithms take the best of both approaches. They are slower than lockset based algorithms but faster than happens-before based algorithms. Similarly, the number of false positives is very small.

There are few works on race detection for SystemC and they are mainly based on static analysis. The tool Scoot [1] has been used to speed up simulations by synthesizing an optimized SystemC scheduler that performs partial-order reduction using information obtained from race analysis. In [17], the authors present a static data race detector that uses gcc plugin as a frontend. Similarly, a static race detector is given in [2] for ESL designs exploiting the dependency among atomic regions in SystemC simulations. This approach suffers from the lack of pointer analysis that leads to false positives.

There are also more general static and dynamic verification approaches for SystemC. Static approaches use model checking and symbolic simulation [3, 14]. However, they are limited in terms of scalability to real designs. Dynamic verification works such as [6, 12, 23] work on the execution trace of the design, they can check correctness of assertions and have better scalability yet lower coverage. In [10], the authors provide the means for the local application of simulation Directed for Exhaustive Simulation (DEC) of schedulings for those parts of the specification where partial-order reduction techniques cannot be applied.

Our goal is to develop a hybrid data race detector for SystemC. Our hybrid solution includes optimizations that do not exist in earlier hybrid approaches [20, 24]. For this purpose, we track causality information in the program to determine concurrent accesses to shared resources. We use a “dynamic binary instrumentation” tool, PIN [15], which does not need the source code of the program, in order to track causality.

3.2 Background

In this section, we describe background on SystemC, vector clocks, and dynamic data race detectors.

3.2.1 Background on SystemC and Vector Clocks

The SystemC scheduler nondeterministically schedules ready-to-run processes and has an asynchronous interleaving semantics where scheduling of processes is non-deterministic. Scheduling of processes occurs at specific locations such as the wait function, or the end of a thread, but not inside the atomic wait-to-wait blocks. The non-deterministic thread scheduling may result in a discrepancy between the simulation and synthesis models. Although scheduling can be restricted with constructs such as explicit events for the lower-level models, for high-level models such as TLM, designers often want the nondeterminism to model nondeterministic choices implicit in the design. The SystemC scheduler is not preemptive; that is, a process runs without interruption until it explicitly gives control back with a wait statement. In [7], the authors show that a nonpreemptive scheduler introduces implicit atomic sections (a wait-to-wait block in a process). Common synchronization objects in SystemC are *sc_event* and *sc_mutex*. *Synchronization operations* such as *sc_event.notify*(), *sc_event.wait*(), *sc_mutex.lock*(), *sc_mutex.unlock*() are executed on these objects.

A partial-order relation, named Lamport's happens-before relation, has commonly been used to track causality in concurrent systems [13]. Lamport's "happens-before" relation ($\rightarrow$) is defined as the smallest transitive relation satisfying the following properties: (a) if actions e and f are generated by the same process, and e occurred before f in real time, then $e \rightarrow f$, and (b) if actions e and f correspond to the send and receive, respectively, of the same message, then $e \rightarrow f$. Two actions e and f are "concurrent" iff $e \not\rightarrow f$ and $f \not\rightarrow e$.

Vector clocks are used to track the happens-before relation [16] during program execution. That is, $e \rightarrow f$ iff $vc_e < vc_f$. Hence, they can be used to determine whether two actions are concurrent. A *vector clock*, vc, is a vector of integers where the size of the vector is determined by the number of processes. Initially, for a process j, $vc_j[i] = 0$, for $i \neq j$, and $vc_j[j] = 1$. In this work, we assign a vector clock to every process. The vector clock is updated after synchronization operations. A process includes a copy of its vector clock in every outgoing message. On receiving a message, it updates its vector clock by taking a component-wise maximum with the vector clock included in the message. Note that happens-before relation has been used in the context of SystemC [6, 9, 12, 23].

3.2.2 Background on Dynamic Race Detectors

There are mainly two types of dynamic race detectors; lockset based detector (LBD) and happens-before based detectors (HBD), which we describe below.

3.2.2.1 Lockset Based Detector (LBD)

Locks are commonly used in concurrent programs for properly synchronizing accesses to shared variables. For example, in SystemC there is an *sc_mutex* class that can be used for locks. A LBD checks that if two threads access a shared variable then they must hold a common lock. Otherwise, the access is not properly synchronized and can lead to a race condition. Eraser [22] is the most well-known LBD.

In the LBD algorithm, every thread and every memory address is associated with a *lockset* variable, which is a set of locks. A thread lockset keeps the set of locks held by the thread and is updated with lock/unlock operations on the lock. A memory address lockset keeps the intersection of locksets of threads that accessed the memory address so far. If any memory address lockset becomes empty, then a race condition warning is given.

Lockset based detectors can be implemented easily and efficiently. However, the disadvantage of these detectors is that they produce many false positives. False positives are disappointing because the programmers waste time examining the outputs of the race detector in race-free situations. The main source of false positives in LBD can be explained as follows. The programmer codes in such a way that there is no common lock held while accessing a shared variable. However, there is an implicit "happens-before" relation in the programmer's mind while coding the accesses. To be more precise, the coder is absolutely sure that one access is going to happen-before the other due to other synchronization operations, hence no race condition can occur. The LBD is unaware of the implicit "happens-before" relation in the coders mind and produces false positives. We show an example of a false positive in the next section.

3.2.2.2 Happens-Before Based Detector (HBD)

This detector checks that if two threads access a shared variable then the accesses are ordered according to the Lamport's happens-before relation described above. Otherwise, the accesses are concurrent and can lead to a race condition. FastTrack and DJIT+ [5] are the most well-known HBDs.

In HBD algorithm, every thread, every memory address, and every synchronization object (*sc_mutex* or *sc_event*) is associated with a vector clock. A thread increases its own component of the vector clock upon releasing a lock, *sc_mutex.unlock*(), or notifying an event, *sc_event.notify*(). When a thread releases a lock then the vector clock of the lock is updated with the thread's vector clock.

When a thread acquires a lock then the vector clock of the thread acquiring the lock is updated with the maximum of the thread's and the lock's vector clocks. This enables the creation of a "happens-before" relation between the two threads due to the release and acquire operations on the lock. Similarly, a "happens-before" relation is created between the threads due to a notify and wait operation on the same SystemC event.

Note that the 'happens-before" relation is generated dynamically during the execution of the model and is tracked by the vector clocks. For example, a happens-before relation is established from any type of notification, say *e.notify*() or *e.notify*(*SC_ZERO_TIME*) to a wait event *wait*(*e*) only if during the execution the thread executing *wait*(*e*) is released by the thread executing the notification on event *e*.

In order to detect race condition on shared memory addresses, for every memory address there is a read and a write vector clock that keeps the vector clock of the last read and write to the address by the thread that accesses the address. We say that a read from a memory address by a thread is race-free provided the read happens after the last write of each thread. This captures read-write type races as there are no read-read races. A write to a memory address by a thread is race-free provided that the write happens after all previous accesses to that variable. This captures write-read or write-write type races.

The advantage of HBD is that it does not produce false positives for the given execution. Whenever a happens-before based race detector produces a warning, there is a thread schedule where two accesses may happen concurrently. The disadvantage of these detectors is that they may miss real races that can occur in an alternative schedule, that is, they generate false negatives. This is because the causality information is dynamically generated only for the observed execution but not for other possible executions. We show an example of a false negative in the next section.

We further specialize the traditional HBD algorithm for SystemC, in that, a race condition is detected when the concurrent access to the shared variable occur in the same delta cycle, as different cycles in SystemC are causally ordered.

Implementing the happens-before relations is not efficient because keeping a vector clock for every thread and every memory address is costly. Furthermore for every memory access such as read or write, there is also a costly vector clock comparison. We next describe our optimized hybrid data race detection solution with low overhead for SystemC programs.

3.3 Our Hybrid Dynamic Race Detection Algorithm

Our hybrid dynamic race detector (Hybrid) is a combination of LBD and HBD algorithms. It consists of two phases, where in Phase 1, a HBD algorithm is run to keep track of concurrent accesses to memory addresses. The difference from the above HBD algorithm is that no happens-before relation is generated for lock

operations, since locks are handled in Phase 2. If a concurrent access is found in Phase 1, then in Phase 2, an LBD algorithm is run to check whether these accesses are protected by a common lock. The most well-known Hybrid detectors for multithreaded programs are [20, 24].

Our algorithm uses the concept of segments whereby it can treat all memory accesses of a thread that hold the same set of locks and that have the same vector clock uniformly leading to efficient implementations. A *segment* [24] is defined as a set of consecutive memory accesses by a single thread. No function calls or synchronization operations are allowed inside the segments but only memory read/write operations, if they exist. Segments of a thread follow a sequential order. When one segment *seg* terminates, the next segment *nseg* starts. A synchronization operation, such as *lock*()/*unlock*() and *notify*()/*wait*(), terminates a segment *seg* and starts a new segment *nseg*. These synchronization operations do not belong to the segment. We observed that when there are few synchronization operations in the application, segments can consist of many lines of source code, making it harder to pinpoint the exact location of a detected race, if the race belongs to that segment. Hence, in order to limit the size of segments, we assume that function calls and returns also start and terminate segments.

A segment belongs to one and only one thread and is associated with a vector clock (which is the vector clock of the thread that it belongs to) and a lockset (set of locks held by the thread at the start of that segment). Furthermore, every memory address is associated with a *read-segment set* and a *write-segment set* denoting respective operations on that address in different segments of different threads.

With segments, we do not keep a separate vector clock (as in HBD) and a separate lockset (as in LBD) for each memory address, the number of which is much higher than the number of segments. Instead, we keep a separate vector clock and a separate lockset for each segment. Similarly, lockset and vector clock comparisons are done in the granularity of segments. These allow us to obtain performance benefits over traditional lockset and happens-before base race detector algorithms.

Our hybrid algorithm is a dynamic race detector that is run during the execution of the given program. In order to implement the algorithm we use a dynamic binary instrumentation tool, PIN [15]. PIN does not require the source code hence can be used on binaries and precompiled IPs. We do not need the debug information in binaries, although the presence of it helps to improve the debugging of data races. Specifically, we instrument memory accesses (read/write operations) and synchronization operations in the given SystemC binary program. During the execution of the instrumented binary program, our algorithm creates and updates segments, read and write segment sets, read and write locksets as well as the corresponding vector clocks.

3.3.1 Algorithm Details

When the instrumented program executes memory access operation E for a shared memory address A, the hybrid race detector runs as shown in Algorithm 3.1. This algorithm has a happens-before based detector phase (Phase 1) and a lockset-based detector phase (Phase 2). The goal is to check whether concurrent accesses on a shared memory address A use common locks or not. If there is no common lock, then a race condition is reported. In our case, the locksets are generated for segments instead of memory addresses. Specifically, in Phase 1 (lines 3–8), we remove all segments that have a happens-before relation to the segment S of memory address A from the read or write segment sets and add S to the corresponding segment set. Hence, the first phase of the algorithm makes sure that the elements of any read-segment set or any write-segment set are pairwise concurrent with each other. This is important because, in Phase 2, comparing locksets of segments that are not concurrent, hence that cannot lead to a race condition, is a waste of resources.

Algorithm 3.1 Hybrid dynamic race detection algorithm (Hybrid)

Input: SystemC binary program P, segment S in thread T, memory address A
Output: race conditions in P

```
    //
    // ———-Phase 1: happens-before phase
 1: let rseg, wseg be read-segment set and write-segment set of memory address A,
    respectively;
 2: let E be a memory access operation on address A in segment S;
 3: if E is read then
 4:     remove all segments that happens-before S from rseg;
 5:     add S to rseg;
 6: else if E is write then
 7:     remove all segments that happens-before S from rseg and wseg;
 8:     add S to wseg;
 9: end if
    //
    // ———-Phase 2: lockset phase
10: for all segment w_i ∈ wseg do
11:     for all segment w_j ∈ wseg do
12:         if write-locksets of w_i, w_j do not have a common lock then
13:             if w_i and w_j occurred in the same delta cycle then
14:                 report RACE CONDITION;
15:             end if
16:         end if
17:     end for
18:     for all segment r_j ∈ rseg do
19:         if write-lockset of w_i and read-lockset r_j do not have a common lock then
20:             if w_i and r_j occurred in the same delta cycle then
21:                 report RACE CONDITION;
22:             end if
23:         end if
24:     end for
25: end for
```

It can be seen from Algorithm that read and write accesses update segment sets differently. On write accesses, both writer and reader segment sets are updated, whereas, on read accesses, only reader segment set is updated. The reason is as follows, on write accesses, it is safe to remove any of the read accesses from *rseg*. Remember that, *rseg* consists of concurrent segments where *A* is read. Since there is no read-read type of data race, removing any segment from *rseg* does not lead to missing any races. On the contrary, on read accesses it is not safe to remove any segment from *wseg*. The reason is that, it may lead to missing a write-write data race because the removed segment might have a potential race with one of the prospective segments in the same set. The outcome of this is that all segments within any segment set is concurrent with each other. However, not all segments in *rseg* are concurrent with all segments in *wseg*, which is handled while checking race among *wseg* and *rseg*.

Then, in Phase 2 (lines 10–21), we check whether for the memory address *A* there can be two concurrent write operations (write-write) or a concurrent write and a read operation (write-read). In both cases we report a race condition warning.

We also implemented several optimizations in our hybrid detector some of which we briefly discuss in this paper. In the first optimization, we observe that maintaining a limited vector clock history cache for the previously calculated vector clock comparisons increase the performance of data race detection. This is motivated by the fact that multiple memory accesses can belong to the same segment and since all these memory accesses have the same vector clock as the segment that they belong to, there may be an excessive number of comparison operations between the same vector clocks. In the second optimization, we define a limit that identifies the maximum number of segments that can be utilized by our approach. This is because discarding segments that are further away from each other can increase the performance as many of the distant segments are unrelated in terms of race detection. In the third optimization, we adopt an approach that detects data races at a rate equal to a user given sampling rate. This approach can make our solution to be applicable in industrially deployed software.

3.3.2 Data Race Detection Examples

We demonstrate our race detection algorithm on two simple SystemC programs. In Fig. 3.2, assuming that during the execution of the program threads are executed in the order (t2, t1, t2), a happens-before relation is constructed from the *notify* in t2 to the *wait* in t1. Our hybrid race detector and the happens-before based detector both correctly return no race on variable *a*, however the lockset based algorithm wrongly returns a race (false positive) on variable *a* as there is no common lock protecting *a*.

Similarly, assuming (t1, t2) execution sequence in Fig. 3.3, a happens-before relation is constructed from the *unlock* in t1 to the *lock* in t2, returning no race condition on *x*. However, in an alternative execution of threads (t2, t1), there is a race on *x*. Our hybrid detector and the lockset detector correctly predict this

```
1  #include <systemc.h>
2
3  SC_MODULE(prog1) {
4
5      sc_event e, f; int a;
6
7      void t1() {
8          a = 2;
9          e.notify();
10     }
11
12     void t2() {
13         wait(e);
14         a = 1;
15     }
16
17     SC_CTOR(prog1) {
18         SC_THREAD(t1);
19         SC_THREAD(t2);
20     }
21 };
22
23
24 int sc_main(int argc, char* argv[]) {
25     prog1 t("prog1");
26     sc_start();
27     return 0;
28 }
```

Fig. 3.2 SystemC Data Race Example 2

race condition from an execution with no races, however the happens-before based detector does not (false negative). Note that one can also come up with an example where the happens-before based detector correctly detects the race, whereas the hybrid detector and the lockset detector do not. In summary, races found by a happens-before or a hybrid detector are a subset of a lockset detector, whereas no such relationship can be said between a happens-before and a hybrid detector.

3.4 Experimental Results

To the best of our knowledge, there is no publicly available race detector for SystemC, hence we implemented all three mentioned detectors using the same framework for a fair comparison. We performed experiments on SystemC and TLM designs from the Accelera distribution as well as an industrial framework. The industrial framework is used for architectural exploration, RTL development, constrained random verification, and early software development. It includes a complete set of BFM and monitor components for several bus protocols including proprietary TLM compliant bus protocols. We used three designs (denoted by ind1, ind2, and ind3) from the framework.

Table 3.1 shows the lines of code (LoC) and the number of potential data races returned by three detectors Hybrid, LBD, and HBD, respectively. The experimental

```
1  #include <systemc.h>
2
3  SC_MODULE(prog2) {
4
5       sc_mutex mutex;
6       int x=0,y=0;
7
8       void t1() {
9            x = 5;
10           mutex.lock();
11           y = 10;
12           mutex.unlock();
13      }
14
15      void t2() {
16           mutex.lock();
17           y = 9;
18           mutex.unlock();
19           x = 10;
20      }
21
22      SC_CTOR(prog2) {
23          SC_THREAD(t1);
24          SC_THREAD(t2);
25      }
26 };
27
28 int sc_main(int argc, char* argv[]) {
29       prog2 t("prog2");
30       sc_start();
31       return 0;
32 }
```

Fig. 3.3 SystemC Data Race Example 3

Table 3.1 The number of potential data races detected by Hybrid, Lockset (LBD), and Happens-before (HBD) Race Detectors

SystemC Design	LoC	Hybrid	LBD	HBD
fir	916	9	13	4
pkt_switch	1045	15	51	8
reset_signal_is	132	12	17	5
rsa	539	0	0	0
lt-specialized_signals	7344	8	39	1
simple_perf	265	3	3	1
simple_fifo	166	1	1	1
at_1_phase	2350	0	0	0
pipe	627	0	0	0
lt-scx_barrier	161	1	1	1
ind1	37,863	30	50	11
ind2	37,950	14	20	12
ind3	37,768	15	51	12

results confirm that the number of reported races is highest with LBD (since it can return false positives) and lowest with HBD (since it can return false negatives). As expected, our hybrid detector combines the best of these two detectors and returns races that cannot be found by the HBD. The slowdowns are 56x, 107x,

and 125x for LBD, Hybrid, and HBD, respectively, which is similar to the results obtained with race detectors in the literature. A reason for the slowdowns is that the overhead of dynamic binary instrumentation is high, which we want to further investigate in the future.

A close examination of data races in the case of industrial designs show that most of the reported races are benign, but there still exist harmful races resulting in corruption of the memory contents, for example in ind1 design. We observe that several SystemC designs are free from data races (rsa, at_1_phase, pipe), which is probably because these designs have been extensively verified. Also, the number of potential data races is higher when the designs are larger as in the case for the industrial designs.

We also performed preliminary experiments using our optimizations. For optimization 1, the execution time decreases 5 % when the history cache size is 1 and the maximum gain is 7 % for a history cache size of 50. For optimization 2, the execution time decreases 24 % when only 4 % of segments (compared to the original number of segments) is kept, yet the number of races remains the same. For optimization 3, the total number of reported races and execution time converge to the sampling rate for almost all applications, which is what we expected to achieve.

3.5 Conclusions and Future Works

We developed a hybrid dynamic race detection algorithm for SystemC/TLM designs with fewer false positives than lockset and fewer false negatives than happens-before algorithms. Our implementation uses dynamic binary instrumentation allowing us to work on designs for which source codes may not be available such as pre-compiled IPs. We implemented optimizations to improve the performance of our hybrid algorithm.

In the future, we plan to expand our experimental works with details of our optimizations as well as investigate approaches to reduce slowdowns. Also, the relationship of our approach to correct-by-construct methodologies for SystemC such as [8, 19] will be investigated.

Acknowledgements This research was supported in part by Bogazici University Research Fund 7223 and the Turkish Academy of Sciences.

References

1. Blanc, N., Kroening, D.: Race analysis for systemc using model checking. ACM Trans. Des. Autom. Electron. Syst. **15**(3), 21:1–21:32 (2010)
2. Chen, W., Han, X., Domer, R.: May-happen-in-parallel analysis based on segment graphs for safe esl models. In: Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE) (2014)

3. Cimatti, A., Narasamdya, I., Roveri, M.: Software model checking systemc. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **32**(5), 774–787 (2013)
4. Flanagan, C., Freund, S.N.: Type-based race detection for java. In: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (2000)
5. Flanagan, C., Freund, S.N.: FastTrack: efficient and precise dynamic race detection. In: Proceedings of the Conference on Programming Language Design and Implementation (2009)
6. Helmstetter, C., Maraninchi, F., Maillet-Contoz, L.: Full simulation coverage for systemC transaction-level models of systems-on-a-chip. Form. Methods Syst. Des. **35**(2), 152–189 (2009)
7. Helmstetter, C., Ponsini, O.: A comparison of two systemC/TLM semantics for formal verification. In: Proceedings of the International Conference on Formal Methods and Models for Co-Design (MEMOCODE) (2008)
8. Herrera, F., Ugarte, I.: Concurrent specification of embedded systems: an insight into the flexibility vs correctness trade-off. In: Tanaka, D.K. (ed.) Embedded Systems - Theory and Design Methodology. InTech (2012)
9. Herrera, F., Villar, E.: A framework for heterogeneous specification and design of electronic embedded systems in systemc. ACM Trans. Des. Autom. Electron. Syst. **12**(3), 22:1–22:31 (2008)
10. Herrera, F., Villar, E.: Local application of simulation directed for exhaustive coverage of schedulings of systemc specifications. In: Proceedings of Forum on Specification Design Languages (FDL) (2009)
11. IEEE Standard for Standard SystemC Language Reference Manual. IEEE Std 1666–2011 (Revision of IEEE Std 1666–2005) pp. 1–638 (2012)
12. Kundu, S., Ganai, M., Gupta, R.: Partial order reduction for scalable testing of systemC TLM designs. In: Proceedings of the 45th Annual Design Automation Conference, pp. 936–941 (2008)
13. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978)
14. Le, H.M., Grosse, D., Herdt, V., Drechsler, R.: Verifying systemc using an intermediate verification language and symbolic simulation. In: Proceedings of the 50th Annual Design Automation Conference (2013)
15. Luk, C.K., Cohn, R.S., Muth, R., Patil, H., Klauser, A., Lowney, P.G., Wallace, S., Reddi, V.J., Hazelwood, K.M.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of Programming Language Design and Implementation (2005)
16. Mattern, F.: Virtual time and global states of distributed systems. In: Proceedings of the Workshop on Distributed Algorithms (WDAG) (1989)
17. Moiseev, M., Glukhikh, M., Zakharov, A., Richter, H.: A static analysis approach to data race detection in systemc designs. In: Proceedings of International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS) (2013)
18. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing heisenbugs in concurrent programs. In: Proceedings of the 8th USENIX Conference on Operating systems Design and Implementation (2008)
19. Niaki, S.H.A., Sander, I.: An automated parallel simulation flow for heterogeneous embedded systems. In: Proceedings of Design, Automation Test in Europe Conference Exhibition (DATE) (2013)
20. O'Callahan, R., Choi, J.D.: Hybrid dynamic data race detection. In: Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2003)
21. Pozniansky, E., Schuster, A.: Multirace: efficient on-the-fly data race detection in multithreaded c++ programs: Research articles. Concurr. Comput. Pract. Exper. **19**(3), 327–340 (2007)

22. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: a dynamic data race detector for multi-threaded programs. In: Proceedings of the 16th ACM Symposium on Operating System Principles (1997)
23. Sen, A.: Concurrency-oriented verification and coverage of system-level designs. ACM Trans. Des. Autom. Electron. Syst. **16**(4), 37:1–37:25 (2011)
24. Serebryany, K., Iskhodzhanov, T.: Threadsanitizer: data race detection in practice. In: Proceedings of the Workshop on Binary Instrumentation and Applications (2009)

Part II
Languages for Requirements

Chapter 4
Semi-formal Representation of Requirements for Automotive Solutions Using SysML

Liana Muşat, Markus Hübl, Andi Buzo, Georg Pelz, Susanne Kandl, and Peter Puschner

Abstract As system complexities are growing with increasing numbers of requirements, the difficulties to manage, process and verify natural language requirements and to keep quality are also increasing. In safety-related applications, as in the automotive domain, this necessity is more pronounced because of the regulations and standards imposed by authorities. Semi-formal representation of requirements is an approach that helps making them more understandable and rigorous.

This chapter deals with semi-formal representation using SysML of two automotive analogue-mixed signal systems, an electronic power switch and an airbag safety circuit. We use diagram-based modelling in order to represent requirements, structure and behaviour, enabling the linking different elements that define the composition and the functionalities of the desired product. We focus on the particular behaviour of such devices and the continuous quantities related to them with emphasis on the two real scenarios.

L. Muşat (✉)
Automotive Power, Infineon Technologies AG, Villach, Austria

Institute of Computer Engineering,Vienna University of Technology, Vienna, Austria
e-mail: Liana.Musat@infineon.com

M. Hübl
Automotive Power, Infineon Technologies AG, Villach, Austria
e-mail: Markus.Huebl@infineon.com

A. Buzo
Methodology development - Automotive, Infineon Technologies Romania&Co SCS, Bucharest, Romania
e-mail: Andi.Buzo@infineon.com

G. Pelz
Methodology development - Automotive, Infineon Technologies AG, Neubiberg, Germany
e-mail: Georg.Pelz@infineon.com

S. Kandl • P. Puschner
Institute of Computer Engineering, Vienna University of Technology, Wien, Austria
e-mail: susanne@vmars.tuwien.ac.at; peter@vmars.tuwien.ac.at

© Springer International Publishing Switzerland 2016

F. Oppenheimer, J.L. Medina Pasaje (eds.), *Languages, Design Methods, and Tools for Electronic System Design*, Lecture Notes in Electrical Engineering 361,
DOI 10.1007/978-3-319-24457-0_4

4.1 Introduction

Over the last decades the number of electrical and electronic (E/E) systems in a car has grown substantially and the trend will be maintained for the next years to satisfy the demand of the market and users for increased automatic control. In parallel, the development complexity and the integration difficulty of E/E systems have also increased due to the high number of requirements that define the automotive sub-systems. Within this context, the automotive community is facing several challenges. First, the management of requirements, i.e. documenting, analysing and tracing of requirements, has become very difficult. Second, there is no explicit association between the description of the requirements in natural language provided by stakeholders and their implementation.

The direct implementation of natural language requirements could raise misunderstanding and misinterpretation which could lead to problems in a later phase of the development project which is considerably more costly than at the beginning. Typical problems are omission of the required features, over-specification or implementation of redundant features caused by a misunderstanding or a lack of completeness and consistency of requirements.

Third, most typical E/E automotive implementations contain elements from the analogue-mixed signal (AMS) domain. While it is relatively straight forward to describe local functionality with natural language, it appears to be more difficult to structure and describe functional relations between various internal and external components. In many existing specification interdependencies between separate blocks are often spread across the document.

In addition, the newly released ISO 26262 standard provides regulations for the development process of automotive systems in order to handle safety-related requirements systematically. The main objective of the standard is to increase the safety by reducing risk of malfunctions that would cause harm to people. This is managed by addressing the whole product lifecycle that supports tailoring of the necessary activities during the lifecycle phases (management, development, production, operation, service and decommission).

Nevertheless, the mentioned challenges can be tackled by more robust requirements through additional formality. The degree of the formalization is debatable though. Fully formalized requirements assure a full definition of the component or the system, becoming objectives for implementation as well as for automatic verification. On the other hand, a formal representation is very complex and requires a deep knowledge of domain, language and tools. Hence the number of developers is restricted to a few experts which are able to deal with the formal requirements.

Semi-formal representation is a reasonable trade-off for adding formality to the requirements without losing the simplicity of the representation. Moreover, it can make the formulation and the presentation of the requirements more intuitive.

One example of standardized semi-formal graphical languages is the System Modelling Language (SysML) [1]. It was developed by the International Council on Systems Engineering's (INCOSE [2]) Model-Driven Systems Design workgroup.

It is dedicated to systems engineering applications (including both hardware and software). SysML is based on the software modelling language UML (Unified Modelling Language), extending the capabilities of UML for the system domain, and excluding specific software parts. The systems' architecture, its behaviour, the requirements and the relationships between all elements of these aspects can be described in SysML. Although semi-formal representations of requirements and SysML in particular are widely used in the software domain, there are fewer attempts in utilizing these for hardware systems or mixed signal components. One explanation for that could be that hardware manufacturers inherited the tradition to describe the specification mainly in natural language and the higher difficulty in representing such systems, especially when it comes to the AMS domain, as most authors for semi-formal modelling in literature make full abstraction of the analogue functionality.

In this work, we present the advantages of a model-based representation of two AMS components from an automotive application. We use SysML to organize the requirements, to specify the interaction of the components with the environment and other components. This study also shows a description of the high-level structure and the behaviour of the components in SysML. The emphasis is put on illustrating the safety mechanisms while there is the possibility for further extension to all requirements. We demonstrate that AMS functionalities and quantities can be successfully modelled with SysML and that such modelling helps in overcoming the above-mentioned issues.

4.2 Related Work

Semi-formal languages are characterized by a well-defined syntax but without unambiguous and precise semantics. They are a good compromise between full formal representations, which can be simulated in a deterministic way, and natural language representations which are vague and tend to be widely misinterpreted in the engineering field due to the diverse backgrounds and experiences among the members of a development team.

Formal methods are mathematics, logic or algebra-based languages used for specification and verification, with well-defined syntax, semantics and rules. They are most popular in the software field. The weaknesses of formal methods are the limits of computational models, the high initial cost for initial implementation and most importantly the usability [3, 4].

On the other hand, natural language description represents the easiest way to capture and communicate the requirements. Nevertheless, with this advantage come many disadvantages for expressing requirements: lack of clarity, amalgamation, confusion and over-flexibility (the same requirement can be expressed in completely different ways).

Semi-formal methods try to inherent advantage from both approaches. They are well-structured and user-friendly; most include graphical notations, others are solely textual based like the Object Constraints Language [5].

One of the widest used and best known semi-formal notations is UML, a standardized general-purpose modelling language in the field of software systems engineering [6]. UML is an object modelling language, and thus it cannot cover all aspects of E/E system development. Architecture Analysis and Design Language (AADL) is another standardized modelling language used for model-based engineering for embedded software system architectures. AADL is a textual modelling language with graphical elements, which addresses application software runtime architecture but excludes the operational environment for the system view [7, 8].

Based on UML, SysML and AADL, an architecture description language specific for automotive embedded systems—EAST-ADL—has been developed and enhanced by several European research projects [9].

Another extension of the UML profile for model-driven development of real time and embedded applications is MARTE (Modelling and Analysis of Real-Time and Embedded systems) [10].

An extended review of several modelling languages including AADL, UML, SysML and MARTE has been conducted by Evensen and Weiss [11]. The criteria for the evaluation in the context of real-time software system applications are scope, formalism and architectural coverage. A comparison summary reveals the limitations of each language. AADL represents an abstraction of real-time operating system components without support for behaviour modelling. UML has no strict formalism and due to the support of a large number of diagrams, a consistent, semantically correct specification is practically hard to maintain. MARTE has a very complex meta-model and it suffers also from the support of large number of diagrams like UML. SysML is primarily targeted for the system engineering domain.

Other semi-formal languages are being developed in various research activities but none of them is widely spread or well-supported by tool vendors. From these only URML—Unified Requirements Modelling Language—is worth mentioning. The most interesting characteristic of URML is the traceability of requirements from different domains—functional requirements, possible threats and hazards or product features [12].

SysML attempts to overcome several of the limitations of the mentioned languages. It represents a visual modelling language that supports specification, analysis, design, verification and validation of a broad range of systems.

SysML, based on a subset of UML, was adapted for system engineering applications, extending UML's capabilities supporting continuous quantities modelling by using parametric diagrams to define performance and quantitative constraints.

This brought the advantage that this kind of representation streamlines communication between heterogeneous teams during the development of the system, as well as the communication to the stakeholders. The requirements are graphically modelled, explicitly representing the mapping and the relationships with each other.

Additionally the system decomposition can be considered in the initial phase of the development activities already.

Although the majority of literature is pointing out the application in software systems, SysML can be used to define models of different domains. Hierarchical modelling of a system and allocating the requirements to the structural and functional elements is one of the most common ways of modelling in SysML [14, 15]. These models are further used as a starting point for Hazard Analysis and FMEA [13]. Other works show the interaction of SysML with Matlab/Simulink for simulations [16]. Moreover SysML can be a base for code generation for System C and C/C++ [17–19]. Further research shows how SysML is transformed or mapped for formal verification [25, 26].

Also, SysML is widely used to support the requirements engineering process due to its specific diagram and relationships for requirements [24, 27–30].

In this research work, we go beyond the state-of-the-art by modelling the requirements for a smart automotive power switch and an airbag IC component from the AMS domain. The analogue effects are not abstracted, but instead they are detailed by the means of activity and state machines diagrams. Then, this behaviour is linked to the requirement and the structural elements. We illustrate these by giving modelling examples of the functionalities for the targeted applications.

4.3 Application To Be Modelled

The methodology was applied on two safety-relevant systems: a protected high-side switch as part of the electric control unit (ECU) that controls the ABS braking system and an IC part of the ECU of the airbag system.

4.3.1 Protected High-Side Switch Description

The hardware device whose requirements we have chosen to model with SysML is a field-effect transistor (MOSFET) equipped with protection and diagnostic functions. It is meant to work as the driver that activates the valve in an anti-lock braking system (ABS), hence the motivation for the presence of protection and diagnostic functions. It represents a typical safety-relevant automotive application which is very sensitive to safety issues. This is why the requirements for this device contain an additional set of explicit safety requirements beside the usual set of requirements. This leads to an increase of the complexity where the semi-formal representation becomes convenient. The following paragraphs give a short overview of the E/E system and its protection and diagnostic functions.

Integrated MOSFETs are widely used for various automotive switching applications, as high-side and low-side switches. High-side switches are power switches that can switch high currents into grounded loads safely. Further high-side switches

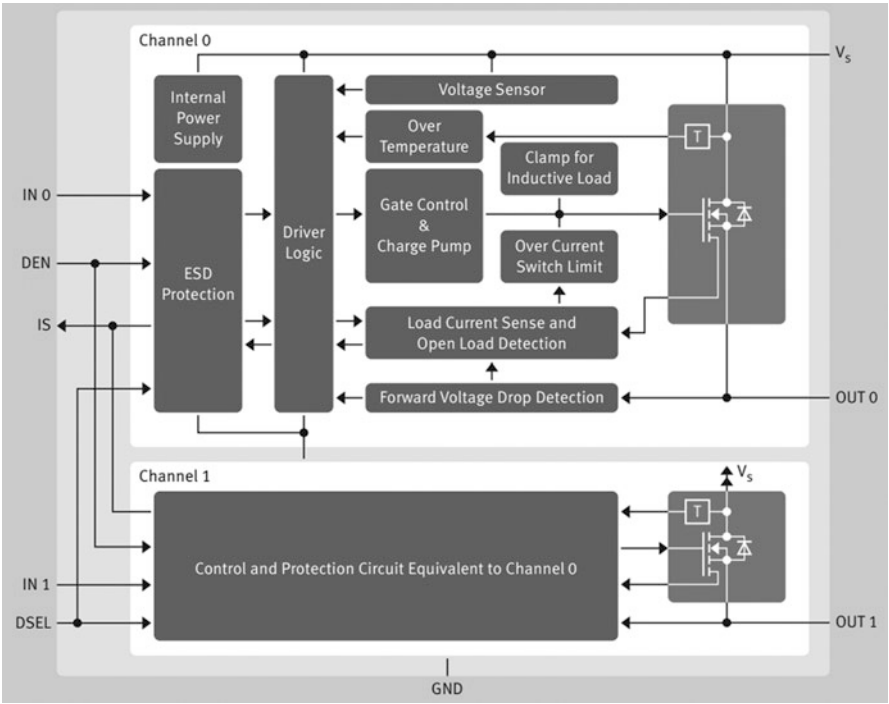


Fig. 4.1 Block diagram of a protected MOSFET [20]

are common in automotive applications due to less wiring and thus reduced system cost. Another advantage of a high-side configuration is the protection of the load due to the disconnection of the supply voltage in OFF state as well as the protected wiring harness [21].

Due to the high demand for safety in the automotive field, high-side switches are designed with built-in protections and diagnostic features. These protection features safeguard that the transistor will not be damaged if one or more of the operating conditions are violated or not fulfilled anymore.

An example of a block diagram for a two-channel smart high-side switch with integrated safety features is illustrated in Fig. 4.1. A large setof available protection and diagnostic features are integrated into a smart high-side switch: electrostatic discharge (ESD) protection, over-voltage and under-voltage protection, reverse battery protection, thermal shutdown protection, over-current protection, short circuit protection, loss of ground or supply voltage protection. These features guarantee that the transistor will not be destroyed or damaged in case one or more of the operating condition requirements are not fulfilled. For example, an over-voltage protection becomes active if the supply voltage is higher than the maximum operating condition and ensures that the high-side switch is not damaged in case of a load dump transient or other high voltage disturbances. In a comparable way,

the under-voltage protection is activated when the supply voltage drops below the operating range. In this case the load is protected by turning off the switch to avoid unpredictable behaviour. Another safety feature is the reverse battery protection, when the supply voltage and ground connections are reversed—which keeps the integrated circuit (IC) and the load safe even when the polarity of the battery is reversed. A current limitation feature is also implemented in order to protect the IC and the load from a short-to-ground failure. The temperature monitoring logic turns off the switch in case of an over-temperature event. The shutdown logic works with a hysteresis to prevent the oscillations of the switching caused by various protections features. The protection techniques are not identical for all devices and differ from model to model and from development organization to development organization [22].

Typical automotive applications for high-side switches are:

- Resistive loads: LEDs, window heating, seat adjustments, auxiliary heating;
- Inductive loads: wipers, ABS and EBS (Electronic Braking System) valves, relays, fan motor and
- Capacitive loads: incandescent lighting and xenon lights [21].

In this application, the high-side switch is part of an ECU, together with a microcontroller (μC), which acts as the master. Some protection functions are fully integrated into the high-side switch, others work only in conjunction with external electronic components. Thus the ECU level has to be taken into consideration for handling protection requirements and functionality of the high-side switch.

The diagnostic function provides feedback to the microcontroller for normal as well as faulty behaviour. The feedback is provided as a current on a sense pin, which is proportional to the respective value on the load.

During faulty conditions the current provided by the diagnostic pin has a well-defined value or a defined range for both ON and OFF states of the transistors as well as for normal functionality and for different fault cases. In ON state, a diagnostic signal can indicate an over-temperature problem, an overload, a partial load loss or an open load. In OFF state it is important to signalize an open load or a short circuit to battery, in order to detect a fault as quickly as possible [22].

The integrated protection functions avoid a destruction of the device as well as provide protection of the connected load, while the diagnostic functions provide information to the μC about the state of the system, e.g. to support protection on the next level and to inform the driver in abnormal conditions. This makes the chosen example system an important asset for safety-related applications, e.g. for controlling an ABS application.

The ABS represents an automotive safety-relevant system used in the braking system of a car. The ECU communicates with the wheel speed sensors and based on data received, the μC controls the valves from the braking system through the high-side switch, preventing the vehicle wheels from locking up and avoiding uncontrolled skidding in critical situations [23]. Valves control represents a vital safety mechanism. A failure in the system, such as when the valve is always closed, the correct braking functionality is affected. An open valve could lead to an

unavailability of the ABS system. Both faulty situations are detected by the system and signalized to the operator of the car. A faulty mechanism can lead to accidents and a loss of life or severe injuries. The protection functions of the high-side switch in the ECU system shall prevent any fault behaviour for the valve control. The diagnostic functions shall report this in order to permit the system to take the correct decision in case of a faulty mechanism [31].

4.3.1.1 Modelling in SysML

Requirements Modelling

The development of complex safety systems requires focus on requirements management from their definition at the beginning of the process until the verification and proof of implementation. The requirements are first described in natural language. This makes them susceptible to incompleteness, inconsistency or ambiguity. Using semi-formal modelling improves the quality of the requirements and of the development process by identifying the issues in the natural description at an earlier phase.

An important feature of SysML is the diagram extension for requirements. This diagram is used to specify the requirements hierarchically and their derivations as well as their relations to other model elements. One of the first steps in product or system definition is the elicitation of requirements.

This step is important for the implementation as well as for verification and validation process. The correct understanding of requirements is the key for the success of the design and the final product.

In SysML, the text-based requirements can be represented graphically in a diagram. This allows the requirements to be expressed hierarchically and linked to requirements or model elements by using different relationships (derive, satisfy, trace, verify, copy or refine). The final list of requirements is often composed of multiple internal and external sources of information. The SysML modelling capabilities offer the possibility to organize stakeholder requirements hierarchically and to group them depending on their specific type or need. These requirements can be refined to internal technical safety requirements (TSRs) by adding any necessary details further. An example of requirement organization in SysML is depicted in Fig. 4.2. It shows only a part of the requirements hierarchy. The continuous lines with crossed end represent a “nesting” relation, i.e. hierarchy relation. The dotted arrows represent the refined relation which exists between the linked requirements. The functional safety requirements (FSRs) from the client are represented in the second row, while on the third row there are represented the internal requirements—built based on the client requirements but enhanced by previous experience and knowledge about implementation. The requirement FSR01, for example—“An open load detection shall be implemented in the electronic control unit”. is refined by two internal TSRs: SR_023 “The protected power switch shall provide for an open load detection in ON mode”. and SR_024 “The protected power switch shall provide for

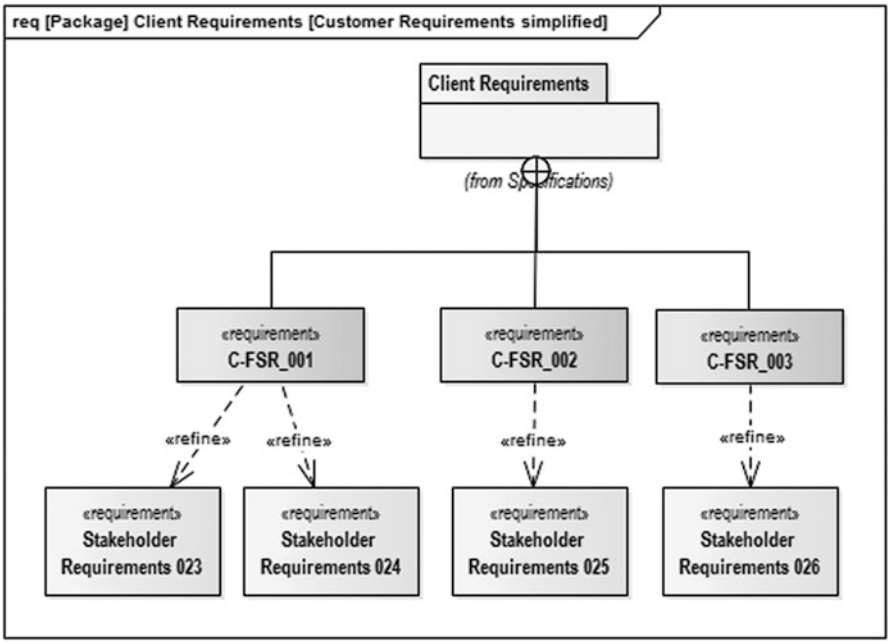


Fig. 4.2 Requirements organization

an open load detection in OFF mode". Every FSR is related to a safety goal, which "Harm through a wire-break between power driver and load shall be avoided". for the example above. This could be modelled and organized in the same way. On the other hand, the internal TSRs can be further refined into a lower abstraction level with implementation details. SR_023 comprises three internal requirements:

- "If the load current Iload is smaller than 100 uA in ON condition, open load shall be detected".
- "Iload shall be measured through the sense current Isense at the sense pin of the power switch, which is a factor of 100 smaller than the load current Iload". and
- "Isense itself shall be measured over the shunt resistance Rshunt using the ADC3 of the microcontroller".

The examples above show that a high-level requirement demands the collaboration of various parts of the ECU and cannot be mapped to a single component. For example, the second internal requirement needs to be split further to become atomic and to be allocated to a component. One part of it "factor 100" is allocated to the power switch, while the other part describes the role of the sense pin, which is allocated to the power switch. It informs the user what the sense pin is supposed to do. A requirement must be split further into sub-requirements as long as it cannot be assigned to a single component (or a single function). In contrast, the third internal requirement is clear and it is allocated to the μC and not to the power switch.

As mentioned, one benefit of modelling the requirements using SysML is the structural and hierarchical organization. This can be easily done directly in SysML or in collaboration with a requirements management tool. It turned out to be especially significant when dealing with a big number of requirements. The graphical representation can be the optimal assistance in the communication with stakeholders, as it gives a clear overview of the requirements. At the same time, refining and linking the external with internal requirements is very beneficial in case of a change request from either side. This creates an easy way to identify which elements are related to the changed or deleted requirements. Additionally, the impact of the modification can be quickly evaluated and the responsible or the related persons are easily identified. Another important advantage of the SysML representation is the possibility to link the requirements with the structural and behavioural models. It offers the possibility to verify that all requirements are covered by the implementation and verification. Due to these aspects, SysML represents a very essential aspect for safety-related requirements, as it provides the proof that the requirements are proposed for implementation and are object to verification—as demanded by the ISO 26262 standard for FSR. Documents and pictures can be attached to the requirement diagrams for implementation proofing, providing a clearer understanding of the requirements depending on the tools capabilities.

Structure Definition

The structure of a system or component can be expressed using SysML block definition diagrams (BDDs) and internal block diagrams (IBDs). The BDD is used to represent the structure using blocks and the interaction between elements or with the environment. The IBD complements the aspects conveyed on BDD, by specifying the internal structure of a single block, including the connections and interfaces between the internal parts.

Figure 4.3 depicts the ECU structure with emphasis on the interaction between the μC and the protected high-side switch. The interfaces are the control input and the diagnostic feedback. The outputs of the high-side switch are used to control the respective load, i.e. the valves used for ABS. In order to achieve a clear overview of the system the battery connections as well as the ground connections are important to be depicted. This model embodies the basic view for introducing the external protection functions, for example, the output protection of the load. The same representation can be provided for all pins and the connections of the switch. Every block diagram can be further refined to an IBD. This provides a transformation from a black box view into a white box view by providing information about the internal structure of the components.

An example of the protected switch is illustrated in Fig. 4.4—it represents a SysML model of the overview depicted in Fig. 4.1. The difference between the two representations is that in SysML the blocks from a diagram are linked to blocks from other diagrams such as the requirements that define them, the system diagrams

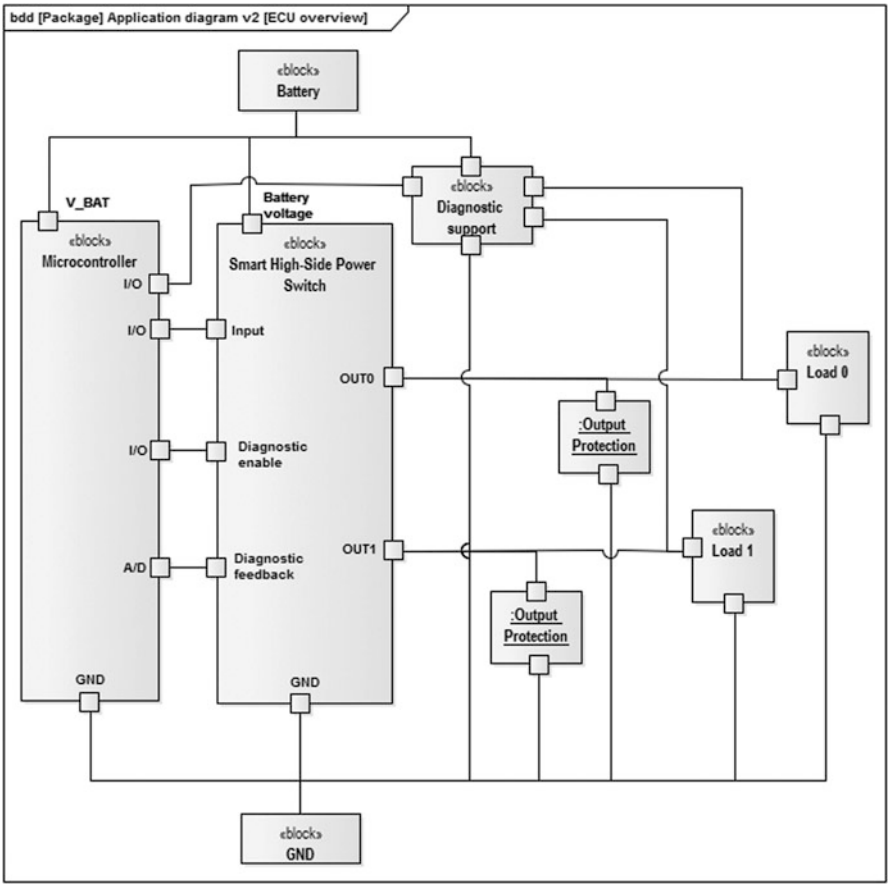


Fig. 4.3 ECU overview

that contain them and the activity or state machine diagrams that describe their characteristics.

The abstraction level used in the BDDs and IBDs is relative and strongly depends on the device being modelled as well as on the people that share this information. Our example, the BDD shown in Fig. 4.3, is addressed to people who do not have strong knowledge, or are not interested in electrical details. It demonstrates the interaction with the exterior devices and includes components that are very sensitive or can play a crucial role in safety-related scenarios. For example, the ground (GND) is specified because it is object to explicit requirements about losing the ground connection or having a short towards it. Instead, Fig. 4.4 provides details about the components of the devices that fulfil different functions specified in the requirements. If requirements raise the necessity to further detail a given component of the high-side switch (e.g. ESD protection block), another IBD will be developed. This IDB will contain the necessary details and it will be linked to the parent

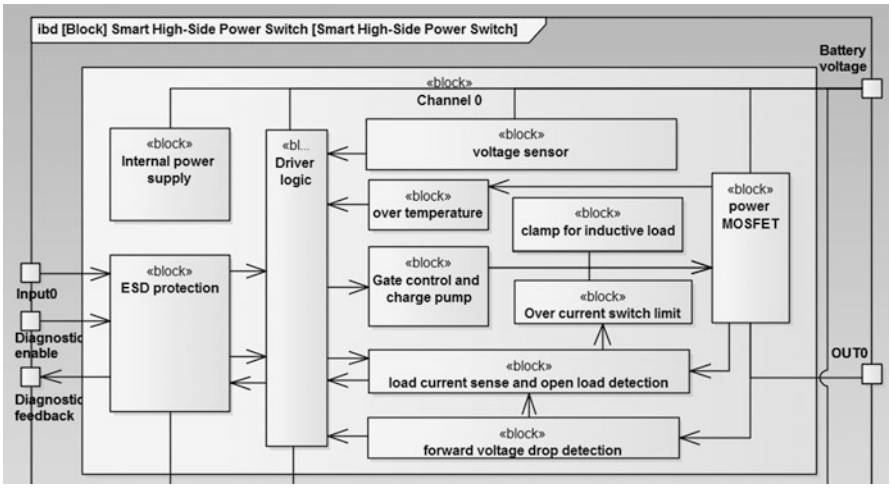


Fig. 4.4 Internal block diagram of the high-side switch

diagram. Hence, a hierarchy of structure diagrams is constructed which allows a fast navigation through different levels of abstraction. Such flexible abstraction level representations have another great advantage: it gives the possibility to hide or show details depending on the confidentiality or the competence that readers have.

In the first stages of a project it is vital to have an overview about the application structure for the high-side switch. Further, the component can be refined using an IBD. The internal representation can be constructed inheriting detailed description of the relevant components, for example, the safety-related elements.

The interaction with the environment and the hierarchical connection, where the component can be represented as a black box as well as with necessary details, prove flexibility and represents a benefit using SysML modelling.

The model remains understandable at each level, by containing only the amount of detail relevant or desired. At the same time all requirements can be linked to the modelled structural elements. In case of a change at the requirement level, the impacted elements are easy to identify. If there are changes later in the design phase, it is fast to identify whether the requirements attached are still fulfilled and implemented.

Behavioural Modelling

Modelling safety protections and diagnostic functions can be achieved by using state machines and activity diagrams. AMS functionality modelling using SysML assumes a clear understanding of the behaviour from a high level of abstraction, which can be further implemented and verified. This means that any uncertainty about the behaviour of the device is solved in advance. If any uncertainty still exists during modelling this will be put in evidence by the modeller and solved by the

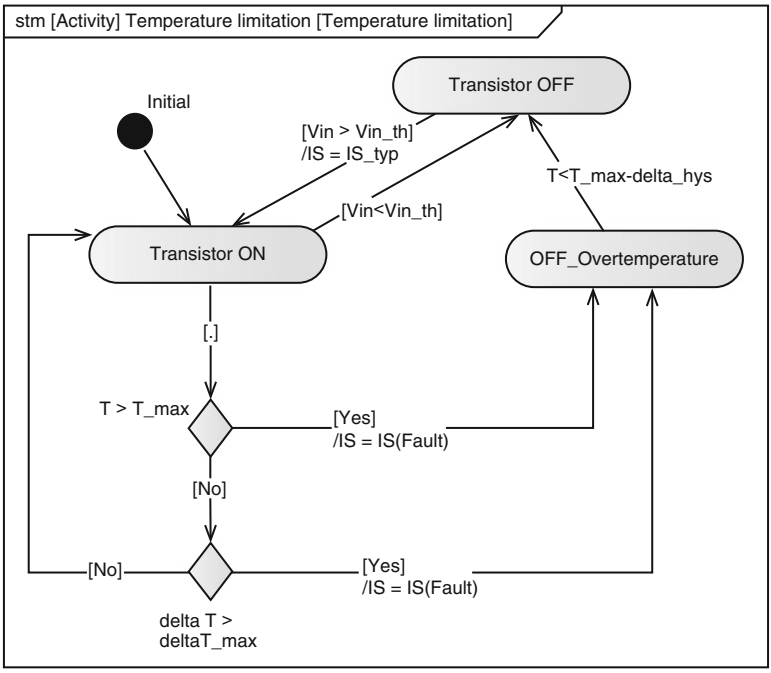


Fig. 4.5 Temperature limitation

interested parts. Therefore, while trying to make the behaviour easy to understand by modelling it, an implicit verification step is performed. With an easy-to-understand model misunderstandings and misinterpretations are easier and faster to find and to correct. There are two approaches used to describe these functionalities. The first one is to model each function separately while the second one models all the functions in the same diagram in order to catch all the aspects related to interactions that may occur among functions.

The description of a single function using state machines consists in modelling an AMS function with states and transitions. For each transition, it is specified the condition of the transition and the diagnostic information—a predefined value for the fault signal. The value of the diagnostic signal is maintained until the mechanism jumps to another state.

Figure 4.5 shows a SysML model of the over-temperature protection function. The behaviour of the transistor is modelled by three states: ON, OFF and OFF_Overtemperature. ON means that the switch is closed and the current flows through the load, while OFF means that the switch is open and the load current is zero. During the ON state the temperature is sensed and compared to a predefined threshold. In case that the temperature rises above a predefined threshold, the protection function reports this to the logic which commands the transition to the OFF_Overtemperature state for the transistor. The same behaviour occurs even

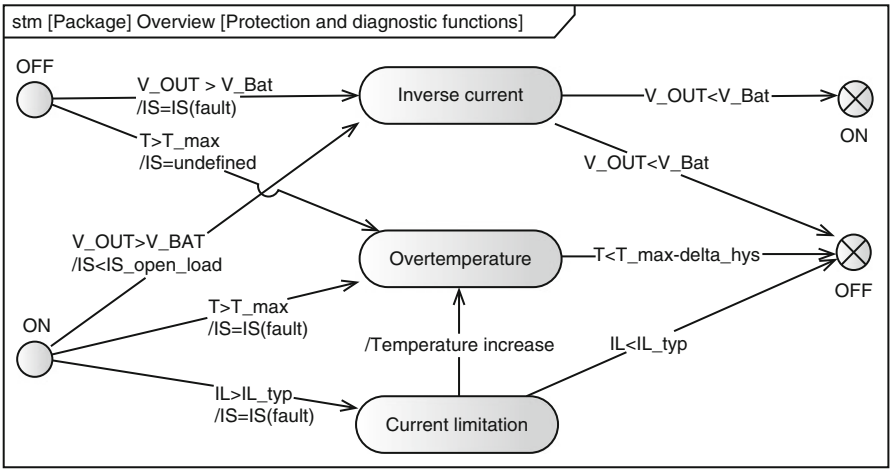


Fig. 4.6 Protection functions

when there is a fast increase in temperature, i.e. when the temperature change (deltaT) within a given interval of time is greater than a threshold (deltaT_max). At the same time the feedback current IS signalizes the faulty case to the uC. The thermal shutdown protects the high-side switch by turning the device completely off. In order to keep the IC from oscillating in and out of the thermal shutdown mode, the hysteresis logic is modelled. In this situation the chip goes back to OFF mode when the temperature value is below the min hysteresis value (delta_hys). A transition back to ON mode is done from OFF mode for the normal voltage conditions (when the input voltage is above the defined threshold to switch from OFF to ON).

The second approach for representing the behaviour is to include all the protection and diagnostic functions into one diagram in order to show how these functions interact and affect each other's behaviour. Figure 4.6 shows an example of three faulty situations and their effect—protection and diagnostic. A fault in the system can happen in both ON and OFF mode. A current limitation can lead to an increase of the temperature and this can lead to thermal shutdown.

Both behavioural models described and illustrated above represent only a small example of the entire behaviour. The complexity of the system, leads also to complex diagrams, where the level of details can be chosen depending on the purpose of the model as well on the target and intention of the person that makes the model.

Every element from the behavioural models (Figs. 4.5 and 4.6), as well as from the structural description (Fig. 4.4) represents a data structure with well-defined properties, like name, author, status, version, complexity, etc. One of the most important properties is represented by the related links, where the relationships between elements are specified. When an element is linked to its requirement, this relation will be available in the properties, containing all the information required:

name of the linked element, element stereotype (e.g. requirement) and connection type (e.g. realization).

A description of all the functionalities together shows how complex the behaviour of the whole device can become. It is relevant to show how the functions work independently as well as how they interact with each other and influence other functions. Compared to natural language description, problems such as an undefined state or transition become visible using state machine or activity diagram representations.

4.3.2 Airbag System

The second use case is also from the automotive domain and it highlights the methodology for the semi-formal modelling approach. An airbag system comprises sensors, actuators and an ECU. The ECU communicates with all sensors and actuators related to the system. Sensors not only include crash detection devices like pressure or acceleration sensors, but also buckle switches and seat occupancy detection; actuators include squibs for airbags as well as safety belt pretensioners.

The ECU receives information about possible crash situations from the sensors and based on internal decision, activates the actuators in case of a crash. The information received by the ECU is available from different types of sensors, like G-force sensors, pressure sensors as well as sensors indicating on which seats persons are present in the vehicle. This information can be either digital or analogue depending on sensor technologies and is transmitted to the ECU. When a crash is sensed, the control unit sends an electrical signal to the corresponding actuators, also known as squibs which will then deploy the airbags.

The main part of the system responsible for the interpretation of the sensors signals and decisions in case of a crash is the ECU. It comprises a microcontroller including the software running on it responsible for the decisional factor in case of a crash, various sensor interfaces, squib drivers, an independent safing engine evaluating sensor data, as well as many other components. For this chapter we assume that all sensor interfaces, squib drivers, and other support functions are integrated into a single component named Airbag IC.

The airbag IC receives the information from the sensors, which is translated to digital signals and communicated to the microcontroller via a Serial Peripherical Interface (SPI) interface. The microcontroller evaluates the received data, and triggers airbag deployment in case a crash is detected based on the received information. As a safety measure, the safing engine also evaluates the sensor data to check for a potential crash independently.

The airbag IC receives the deployment request from the microcontroller, confirmed by the safing engine and releases the squibs for airbags as well as for seatbelt pretensioners.

The combination and interaction of the hardware elements, both analogue and digital, inside of the airbag IC or the safing engine as well as its communication

with the microcontroller on the ECU and with the sensors and actuators outside of the ECU as well as the safety relevance of the entire application considerably increases the complexity in all development phases of the product.

As for the protected high-side switch, the requirements specification for the airbag IC and the safing engine includes and highlights the safety requirements, mapped to diagnostic functions and bidirectional confirmation or rejection of crash detection between the microcontroller and the safing engine.

4.3.2.1 Modelling in SysML

Requirements Modelling

The requirement specification documents have been defined at the level of the airbag IC as well as at the level of the internal sub-systems. Due to the high complexity of the whole system we have selected a sub-system for analysis and modelling, namely the safing engine, motivated by its safety relevance.

The first step is the organization of the natural language requirements in models, depending on their types and relationships. Figure 4.7 depicts the top-level hierarchy of the requirements, classified in different categories: FSRs, normal operation requirements, implementation requirements, requirements coming from standards, chip package requirements and application information. The requirements are further split into sub-categories in order to ease for implementation, verification, validation and back-tracing from the results.

Organizing requirements into packages that correspond to various categories and stakeholders provides consistency with the specification document where the requirements are defined in natural language; it facilitates the configuration change management processes and offers support in organizing the further verification of requirements.

The FSRs include three general categories: the TSRs, the safety requirements related to architecture (for both software and hardware) and the safety-relevant use cases. The set of safety requirements presents all “requirements for the safety instrumented functions that have to be performed by the safety instrumented systems” (definition according to the standard IEC 61511). For example, we can consider an analog to digital converter (ADC) as a safety-critical architectural element. In order to fulfil this safety requirement, one possibility is to implement a second ADC as a redundancy mechanism, in order to mask any faulty behaviour of the converter. Another example for a safety architectural element is a supply voltage regulator for a digital logic circuit. A fail of the regulator may affect the computational logic. As a safety measure, under- and over-voltage mechanisms need to be implemented so that they will detect a fluctuation of the supply voltage and prevent the circuit from being damaged. The under- and over-voltage detection mechanisms should be supplied by different sources in order not to be affected by the same problem as the supply voltage regulator.

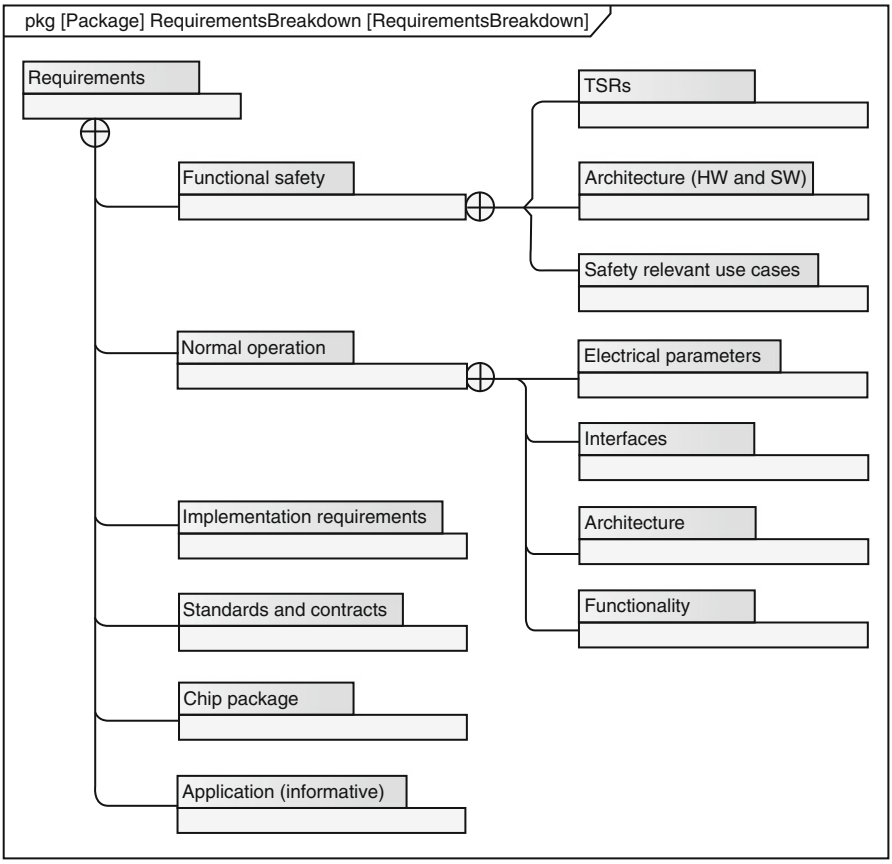


Fig. 4.7 Requirements breakdown for the Airbag Safing Engine

Another brake down of requirements is done for normal operation, based on application needs and specifics such as mandatory electrical parameters, information for the interfaces to the outside environment, architectural and functional requirements. The electrical parameters are usually listed in a table, with the minimum, typical and maximum values that the product needs to fulfil under certain defined operating conditions (e.g. for a given temperature range). A lot of electrical parameters are related to the safety and can be traced to the related safety requirements. As presented in the example above, the limit values for the under- and over-voltage will be specified in the table for parametric requirements.

Interface requirements shall include all the information necessary for the interaction with the outside world, i.e. what are the inputs and outputs, which types of inputs they are and what kind of information is communicated through these interfaces. For example, for the definition of a SPI communication, there are specific

inputs and outputs well defined in the SPI standard and detailed depending on the application.

The architecture requirements represent a high-level view of the targeted structure—usually centred on the functionality—and will differ from the real implementation.

The functionality requirements describe all the needs for the behaviour of the system. They include, for example, operating modes (like unpowered, normal operation, safe mode, sleep mode, etc.), combinatorial and sequential behaviour (illustrated sometimes by waveforms drawings) or configuration options.

The requirements database can also include design and manufacturing requirements, compiled as implementation requirements, which are important for development and production of the device.

Information and requirements for the IC package must also be added, as well as requirements from standards that need to be fulfilled by the product. For example, EMC (Electro-Magnetic Compatibility) standards usually provide information for dedicated measurements during verification.

We suggest adding information about the use of the targeted product as application notes in a separate container. Unlike to other requirements, the notes are only informative but support a human reader in understanding the normative requirements and their motivation. Although tracing or verification of information categorized as Application (informative) is not needed, it's recommended to use a common requirements database to ease use and maintenance of all information. Application notes may help to do verification setups properly and—although not an explicit target of requirements modelling here—support the validation of requirements. Trace links between normative requirements and informative notes may be helpful.

The SysML package structure is used to create a requirements portioning model, which is useful for identifying a complete set of requirements, an essential part in the requirements definition. Each specification package contains the text-based requirements for that category. Relationships between requirements from different packages can be defined, for example by explicitly linking requirements from a safety standard (e.g. ISO26262) to the FSRs. The package is organized using a containment relationship, the representation allowing a compact way to view the hierarchical decomposition.

The requirements are compiled from different sources owned by different stakeholders. The structural decomposition of requirements categories is done using BDD, while the requirements for each category are defined using requirements diagrams.

Figure 4.8 depicts an example of requirements for functionalities that are suggested to be implemented as a state machine. The safing engine is required to work in four different modes: startup mode, diagnostic mode, normal mode and safe mode. The behaviour of each mode as well as the transitions between the modes are described in natural language. Startup represents a safe state where the configuration for the safing engine is done. In diagnostic mode checks of the entire functionality of the safing engine are performed to ensure proper functionality. The normal mode represents the normal working mode, where background checking is still performed.

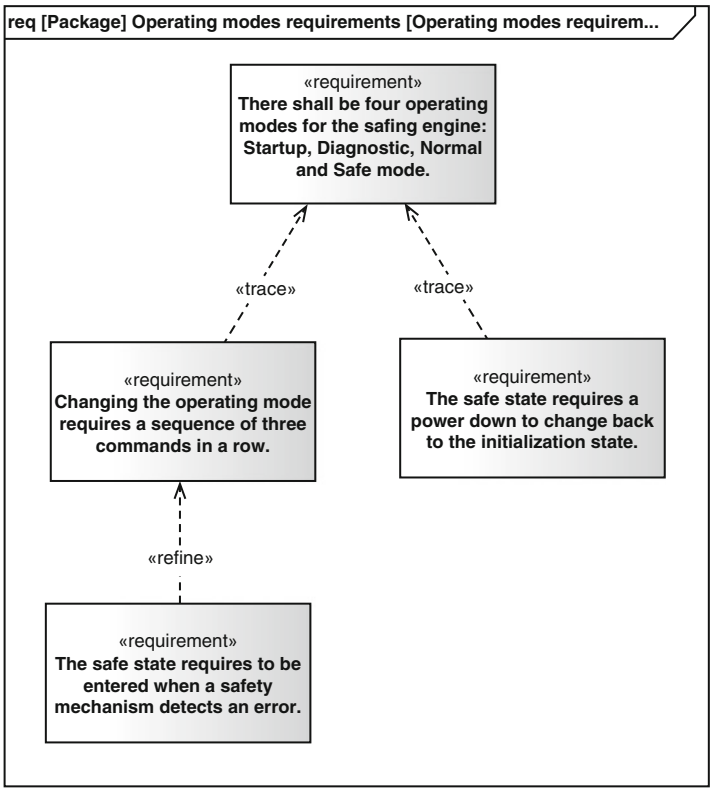


Fig. 4.8 Safing engine requirements example

If a fault is detected at any state, the safing engine will change to the safe mode, which can be exited only by a power down and power up back to the initialization.

There are three relationships types that are used to show how one requirement is related to another: nesting, derive and copy. And there are three types of relationships that can be used to connect a requirement to any type of model element, not only another requirement. These are satisfy, refine and trace relationships. In the context above we use the refine and trace relationships to show the relations between the requirements. These various types of relationships highlight explicitly the connections between different parts of a model as a way to maintain the consistency of the model. If a requirement is not traced to a model element for structure or behaviour, it should be checked weather the requirement is necessary or weather the model element is not yet available. Capturing the traceability within a SysML model enables the possibility to perform a downstream impact analysis. A behavioural or structural model can be easily identified when it depends on a requirement that has changed over time, an advantage of reducing the time and costs when implementing changes in a design over the system development cycle.

Structure Definition

Once the requirements are available in the modelling environment, the second step is the construction of the structural view of the system using BDDs and IBDs based on the natural language requirements. The main purpose is to reflect only the information described in natural language, without crossing the border towards design or implementation. For this there are three views taken into consideration: the context of the system—the application and interaction with the environment, the system with its inputs and outputs—analogue and digital pins, and the internal description—based on the desired level of detail described in the natural language requirements.

Figure 4.9 depicts the main ECU blocks embedded in the application context for the safing engine. The microcontroller as well as the safing engine receive information from the sensors via the sensor interface module, as well the information if the seat is occupied and if the seat occupant has the seat belt buckled or not. In case the microcontroller detects a crash based on the sensor data received and the safing engine confirms the crash, it will send a fire command to the squib drivers. When the safing engine detects a crash situation—also based on the sensor data received—and gets a confirmation from the microcontroller, it will enable the power supply for the squib drivers. Only when both entities detect a crash and confirm this situation to each other the airbag will be deployed during the time indicated by the requirements.

The model shows a high-level representation of the structure based on the requirements. When further refining the architecture, blocks as well as interfaces between blocks may change, e.g. to improve implementation efficiency by using blocks and interfaces as shared resources or to split blocks and interfaces as safety mechanisms.

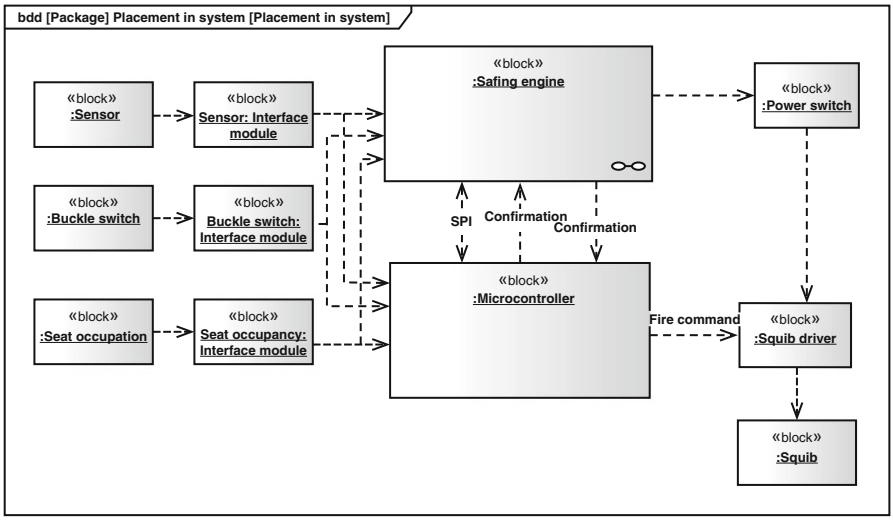


Fig. 4.9 Safing engine system interaction

The safing engine is further modelled based on the structural requirements using IBDs while the structure elements are linked to their requirements.

The main components of the safing engine are depicted as blocks. In order to depict the interconnections and communication flow between the blocks, dependency relationships are used. The diagram shows how sensor data passes through processing blocks to perform computations for crash detection. The blocks are further hierarchically decomposed as far as necessary to provide proper targets for explicit linking to requirements.

The majority of the blocks are defined in component libraries and then instantiated in the structural model view. The main aim of this representation is to provide a clear and simple view of the safing engine interactions with outside elements and the type of interactions and further to describe the decomposition and the relations between the internal components.

Behavioural Description

The requirements listed in section A represent a short example of the functional requirements of the operating modes of the safing engine. The functionality expressed by the requirements is modelled using a state diagram. It comprises four modes: initialization mode, diagnostic mode, normal mode and safe mode. The four modes can be modelled using state blocks in a state diagram (as in the example illustrated in Fig. 4.10) and further used as a state machine element in the next higher level of the model hierarchy.

The power up of the device has been represented by as an entry point item. When the device is powered up, it enters automatically in the initialization state. A transition from initialization mode to diagnostic mode will be performed only by receiving three specific commands in a row, which are described separately. The same applies for the change from the diagnostic mode to normal mode. The safe mode will be reached as a result of a fault detection in any of the other modes or when a specific unlock sequence for entering the safe mode is received from the microcontroller. The natural language requirement states that the safe mode can be exited only by a power down of the system. In fact any state can be exit by a power down, but the specific description of these requirements for the safe mode is intended as a verification requirement explicitly described.

The main issue detected when modelling the requirements using the state diagram, was the incompleteness of natural language description of the transitions from one state to another by a specific sequence of three commands—the requirement "Changing the operating mode requires a sequence of three commands in a row". The order of the commands as well as specifying which commands are needed is described separately in the requirements for the communications.

It was not specified in natural language requirements how to react in case of wrong commands or wrong command sequence. Without a specification for these cases it is unclear whether the system shall keep the current state or enter safe mode.

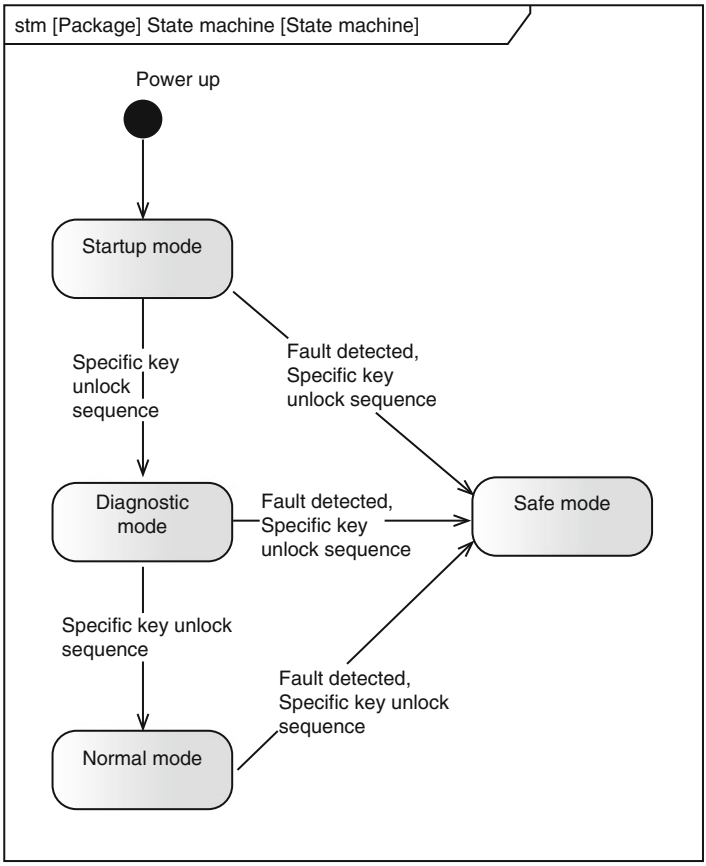


Fig. 4.10 Safing engine state machine functionality

Natural language requirement descriptions are commonly complemented by drawings reflecting structure or behaviour. Typical examples for AMS designs are block diagrams, state diagrams, truth tables, waveform diagrams and even transistor level circuit illustrations. For most of these drawings we recommend to use SysML to benefit from the standardized syntax and semantics. Only those illustrations that have no direct representation in SysML and cannot be translated to SysML without losing significance and clarity should be left in their original form. But they still should be captured by the modelling environment to allow consistent linking between requirements and other SysML objects.

4.4 Conclusions

The diagram-based representation of the requirements allows a better organization, processing and classification of the requirements compared to classical representations like natural language text. By exploiting the SysML capabilities the requirements can be linked to each other according to the relation that exists between two requirements. Moreover the requirements can be linked to structure elements of the device and behaviour models. Such links facilitate the impact analysis of the requirements and guide changes that can occur in different phases of the development.

The diagrams used for modelling the structure of the overall system and the internal structure of components permit a flexible level of abstraction. This flexibility allows hiding or revealing details depending on complexity of the device, the knowledge and expertise of the involved people and the level of information confidentiality.

These features were illustrated by modelling two AMS hardware components: a protected power switch and a safety-relevant module of an airbag system—a safing engine related to the decision to fire squibs for the airbag deployment.

For both SysML use-case examples—the protected power switch and the safing engine—the behaviours of the protection functions and the entire activity, respectively, were modelled by using state machines. State machines offer the possibility to reflect the specifications of events that trigger a given functionality and the actions that are taken once the trigger conditions are fulfilled. All types of diagrams (behavioural, structural and requirements) were linked to each other in order to ensure the integrity of the requirements set, as illustrated in the sections before.

By creating a semi-formal model ambiguities are recognized easier and avoided compared to natural language. The explicit allocation of requirements to blocks or sub-systems encourage or even require to perform a basic feasibility check and hence improve the quality of the specification.

Another possible usage of this model is to connect it to concept and design models which can be simulated and verified. The link between representations as well as requirements needs to be maintained. Furthermore, by using SysML it is possible to specify explicitly test cases that are not deduced from other requirements and link them to different parts in the model that will be refined further and used for verification, especially to ensure that safety requirements are fulfilled.

Acknowledgments The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement Nr. 295311 and the Austrian Research Promotion Agency FFG under the program "Forschung, Innovation und Technologie für Informationstechnologien (FIT-IT)".

References

1. OMG System Modeling Languages. http://www.omgsysml.org/. 13 Feb 2014 (01 May 2015)
2. International Council of Systems Engineering. http://www.incose.org/. 02 Apr 2014 (01 May 2015)
3. Collins, M.: Dependable Embedded Systems, Topic: "Formal Methods." Carnegie Mellon University (1998). https://www.ece.cmu.edu/~koopman/des_s99/formal_methods/
4. Broadfoot, G.H.: ASD case notes: Costs and benefits of applying formal methods to industrial control software. In: FM 2005: Formal Methods. LNCS, vol. 3582, pp. 548–551. Springer (2005)
5. Eltahir, S., Musa, M.: On practicality of using integrated semi-formal modeling tools. In: The International Arab Conference on Information Technology, 2008
6. Unified Modeling Language (UML). http://www.uml.org/. 14 Feb 2014 (01 May 2015)
7. Feiler, P.H., Gluch, D.O.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language. Addison-Wesley Professional (2012)
8. Architecture Analysis & Design Language (AADL). http://www.aadl.info/aadl/currentsite/ (2012) (01 May 2015)
9. EAST-ADL. http://www.east-adl.info/ (2014) (05 May 2014)
10. Modeling and Analysis of Real-time and Embedded systems (MARTE). http://www.omg.org/omgmarte/Tutorial.htm (2008) (01 May 2015)
11. Evensen, K.D., Weiss, K.A.: A comparison and evaluation of real-time software systems modeling languages. Presented at the Aerospace Conference, Georgia, Atlanta, 2010
12. Helming, J., Koegel, M., Schneider, M., Haeger, M., Kaminski, C., Bruegge, B., Berenbach, B.: Towards a unified requirements modeling language. In: Fifth International Workshop on Requirements Engineering Visualization (REV), 2010, pp. 53–57
13. Kaiser, B., Klaas, V., Schulz, S., Herbst, C., Lascych, P.: Integrating system modelling with safety activities. In: SAFECOMP Proceedings, 2010
14. Adedjouma M., Dubois, H., Maaziz, K., Terrier, F.: A model-driven requirement engineering process compliant with automotive domain standards. In: Model Driven Tool and Process Integration, 2010
15. Sontos, M.d., Vrancken, J.: Model-driven user requirements specification using SysML. J. Software **3**, 57–68 (2008)
16. Wang, B., Baras, J.S.: Model-based design framework for wireless sensor networks using SysML, Simulink and Modelica. https://www.src.org/library/publication/p061828/. 28 Oct 2011
17. Mueller, W., He Da, Mischkalla, F., Wegele, A., Whiston, P., Peñil, P., Villar, E., Mitas, N., Kritharidis, D., Azcarate, F., Carballeda, M.: The SATURN approach to SysML-Based HW/SW Codesign. In: IEEE Computer Society Annual Symposium on VLSI (ISVLSI), 2010, pp. 506–511
18. Mischkalla, F., He Da, Mueller, W.: Code generation for QEMU/SystemC Cosimulation from Cosimulation from SysML. Presented at the MeCoES Workshop, 2012
19. Waseem, R.: Accelerating High-Level SysML and SystemC SoC Designs. http://www.design-reuse.com/articles/17562/high-level-sysml-systemc-soc-designs.html (01.06.2015)
20. Infineon Technologies AG: Product_Brief: BTT6050-2EKA (Truck device) PROFET™ + 24V. http://www.infineon.com/dgdl?folderId=db3a30431400ef68011421b54e2e0564&fileId=db3a30433784a0400137984c3a63271b&intc=0120035 (01.06.2015)
21. Infineon Technologies AG: Introduction to PROFET™. http://www.infineon.com/dgdl/Introduction+to+PROFET%E2%84%A2.pdf?folderId=db3a30431400ef68011421b54e2e0564&fileId=db3a304332ae7b090132b527d9173083 (01.06.2015)
22. Infineon Technologies AG: Protected high side drivers. In: Bridging Theory into Practice – Fundamentals of Power Semiconductors for Automotive Applications, 2nd edn. pp. 125–149. Infineon Technologies AG, Munich (2008)

23. Burton, D., Delaney, A., Newstead, S., Logan, D., Fildes, B.: Effectiveness of ABS and vehicle stability control systems. RACV Research Report (April, 2004). http://www.monash.edu.au/miri/research/reports/other/racv-abs-braking-system-effectiveness.pdf (24 Apr 2014)
24. Petin, J.-F., Evrot, D., Morel, G., Lamy, P.: Combining SysML and formal models for safety requirements verification. In: International Conference on software & Systems Engineering and their Application, 2010
25. Bryans, J., Payne, R., Holt, J., Perry, S.: Semi-formal and formal interface specification for system of systems architecture. In: IEEE International Systems Conference (SysCon), 2013, pp. 612–619
26. Jarraya, Y., Soeanu, A., Debbabi, M., Hassaine, F.: 10-Automatic verification and performance analysis of time-constrained SysML activity diagrams. In: IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, 2007, pp. 515–522
27. Gnaho, C., Semmak, F., Belkaid, Y., Laleau, R.: Goal-based requirements engineering in topcased environment. Presented at the TopCased Days, Paris, France, 2011. http://gforge.enseeiht.fr/docman/view.php/52/4276/A1-LACL.pdf (05.05.2014)
28. Hove, D., Goknil, A., Kurtev, I., Berg, K., Goede, K.: Change impact analysis for SysML requirements models based on semantics of trace relations. Presented at the ECMDA Traceability Workshop, Enschede, Netherlands, 2009
29. Favaro, J., Koning, H.-P., Schreiner, R., Olive, X.: Next generation requirements engineering. http://www.intecs.it/PDF/NextGenRE_INCOSE_FINAL_2012.pdf (01.06.2015)
30. Bachhuber, A.: Requirements engineering in the product life cycle of continental automotive. Presented at the RFConf, Munich, German, 2013. http://www.hood-group.com/fileadmin/project/reconf/VortraegePDF/mm3_bachhuber_requirements_engineering_in_the_product_life_cycle_of_continental_automotive.pdf (01.06.2015)
31. Infineon Technologies AG: Smart high side switch. http://www.infineon.com/profet (01.06.2015)

Chapter 5
A New Property Language for the Specification of Hardware-Dependent Embedded System Software

Binghao Bao, Carlos Villarraga, Bernard Schmidt, Dominik Stoffel, and Wolfgang Kunz

Abstract This work introduces a new property language for describing the behaviour of low-level hardware-dependent software. The design of the language is motivated by the industrial success of property languages for hardware verification by simulation and formal techniques. The new language is constructed to concisely capture the timed behaviour of the interactions between software and hardware by means of sequences. In this chapter we present how the proposed verification language can be used to perform formal verification based on a computational model called *program netlist*. We show how the sequence model of the language is synthesised and combined with the program netlist so that a unified formula for a decision procedure, e.g., a SAT solver, can be constructed. Furthermore, a method for coverage analysis of property sets is introduced. The coverage criterion we propose determines whether or not the property set completely describes the input/output functional behaviour of a program. The work presents a case study showing how to use the proposed property language in order to specify an industrial implementation of a LIN (Local Interconnect Network) bus driver.

5.1 Introduction

Besides continuous advances in methods and algorithms for formal property checking of hardware designs, also the languages for formulating properties have played an important role for the adoption of formal verification techniques in industry in the last years. For instance, SystemVerilog Assertions (SVA) [15] and Property Specification Language (PSL) [1] allow to concisely specify the behaviour of the hardware, which is typically described at register transfer level (RTL).

B. Bao • C. Villarraga (✉) • B. Schmidt • D. Stoffel • W. Kunz
University of Kaiserslautern, Kaiserlautern, Germany
e-mail: bao@eit.uni-kl.de; villarraga@eit.uni-kl.de; schmidt@eit.uni-kl.de; stoffel@eit.uni-kl.de; kunz@eit.uni-kl.de

© Springer International Publishing Switzerland 2016

F. Oppenheimer, J.L. Medina Pasaje (eds.), *Languages, Design Methods, and Tools for Electronic System Design*, Lecture Notes in Electrical Engineering 361,
DOI 10.1007/978-3-319-24457-0_5

While being founded in a strictly defined mathematical framework, these property languages include various syntactic enhancements offering a natural and easy way to capture temporal behaviours of the design. Current commercial technology allows for checking assertions using simulation or formal verification engines.

On the other side, for the case of embedded software (SW) there is an increasing necessity of integrating formal verification also to the verification flows used in industry. In this work we focus on the verification of hardware-dependent software which is the part of the software in an embedded system that interacts directly with the surrounding hardware (HW). There are a number of reasons why we focus on this kind of software. Hardware-dependent software is a critical component in embedded systems since all other software layers (e.g., the operating system, application software, etc.) are built on top of it. Additionally, hardware-dependent software in embedded systems performs control-intensive tasks with complex interactions with the hardware and with other software layers, making development error prone and systems difficult to test. Because of the reactive behaviour of HW/SW interaction, specification languages and validation methods as they have been developed for application-level software are in many cases not suitable. This work proposes a new property language facilitating the specification of hardware-dependent software behaviour in embedded systems. Similar to property languages used for hardware, this new language allows to capture the reactive behaviour of the hardware-dependent software by using sequences describing series of input/output operations performed by the software at its interfaces.

The property language proposed here can be employed for simulation or verification purposes. However, in this work we present particularly how this language can be used in conjunction with a computational model called *program netlist* [14] in order to perform formal property checking. A program netlist is a combinational Boolean model representing compactly all possible execution paths of a software program. It is built using hardware models of the machine instructions executed in the program, and is therefore suitable for representing hardware-dependent software. For generating a program netlist, path-oriented techniques related to *symbolic execution* [5] are used. The actual flow of program control is modelled by additional logic added to the program netlist that enhances the efficiency of SAT reasoning on program segments and entire paths.

Unlike methods based on symbolic execution in which properties are proven by traversing explicitly the possible execution paths of a given program, in this work, we adopt the approach of [14] which employs a SAT solver in order to perform this traversal. A SAT proof benefits from the control logic in the program netlist by being able to focus on the execution paths being important for the particular problem instance and to prune at once entire execution paths that are not relevant. The effectiveness of this approach has been shown in [14].

In order to use a SAT solver for path traversal it is necessary to create a combined model containing the logic for the property and the program netlist such that the SAT solver has “the global view on the verification problem” (instead of having only the view of the problem for individual execution paths). For the global view, a model of the input/output sequences of the software is synthesised and integrated to the model.

Since formal verification examines every possible input scenario, the usual coverage criteria evaluating the quality of test cases for software are not suitable for formal property checking. In case of software property checking, verification engineers face the same problem as engineers in hardware verification: "Does my property set cover every aspect of the design?" A number of methods for coverage analysis attempting to prove that a property set is "complete" have been successfully applied to hardware designs [4, 6, 9, 11]. In these approaches a set of safety properties is called a *complete specification* or simply *complete* if it uniquely describes the behaviour of a design. More precisely, these methods prove the completeness of a property set by means of checking to what extent the property set *uniquely* specifies the input and output behaviour of a hardware design. Based on these results, in this work we develop a method for proving the completeness of software properties specified in the software property language presented in this chapter. Such a completeness check is of particular importance for software property sets because a complete set of properties can, at least in principle, fully replace classical software tests. A typical source of error when writing software is that the programmer simply forgets to treat certain input sequences in his program, causing undefined behaviour when these inputs occur at runtime. Also, a verification engineer may forget to specify tests (or, in our case, properties) for such missing input sequences so that the bug can escape verification. The completeness check presented in this work removes such verification gaps. Because of similarity between hardware and low-level software, our method for proving the completeness of software property sets checks whether every input/output sequence is specified by a property in the set. The checking algorithm leverages the property language presented in this chapter, which allows for referring to the interfaces of the software program in order to describe its reactive and sequential behaviour.

As of today, there are a number of different approaches for formalising properties of embedded software. *Run-time assertions* are being used widely for testing [16] and formal verification [7, 19], of embedded software. For example, high-level programming languages like C provide the *assert()* statement for specifying predicates over the values of program variables. The main use of run-time assertions is to describe properties that are valid locally. More specifically, this kind of property is evaluated only when a program run reaches the location where the involved *assert()* statement has been placed. While run-time assertions have the advantage that the user is not required to learn a new language in order to specify properties, their main limitation is in what can be expressed. For the case of application software or for simple transformational code run-time assertions may be sufficient; however, for hardware-dependent software it is necessary to be able to describe reactive behaviour, relating to the inputs, outputs and states of the software and hardware *at different points in time*. For specifying temporal behaviour, temporal logics such as CTL and LTL [8, 12] can be used. There exist verification tools such as [10, 13] that accept temporal formulas directly as PSL. However, although CTL and LTL are powerful in formulating temporal relationships, they are hard to understand and use in practice.

Other tools such as [2, 3], in a similar way, employ automata in order to temporal specify properties. The use of automata can be convenient in many cases since they are easier to understand by a designer or verification engineer than temporal formulas in CTL or LTL. However, except for simple cases, the process of modelling a property using an automaton is cumbersome and error prone.

Different to the approaches mentioned previously, in this work we present a new verification language for hardware-dependent embedded software that allows to specify the temporal behaviour of interactions between software and hardware. The proposed language is intuitive and easy to use for the verification engineer. It allows to refer to the interfaces of the software and to describe explicitly the sequences of input/output operations at these interfaces. It adopts many syntactic elements from the C language, which makes learning the new language easy for software engineers. To the best of our knowledge, this is the first work on a property language for hardware-dependent embedded software with the characteristics mentioned above.

The remainder of the chapter is organised as follows. Section 5.2 reviews the computational model used in this work. The software PSL is described in Sect. 5.3, and it is applied to specifying properties for a LIN driver implemented in software, as presented in Sect. 5.5. In addition, a method for evaluating the completeness of property sets is described in Sect. 5.4. A conclusion and outline of future work are given in Sect. 5.6.

5.2 Low-Level Software Model

In this work we show how the language to be presented in Sect. 5.3 can be used together with a model for hardware-dependent low-level software in embedded systems, called *program netlist*, in order to perform formal property checking. We first review basic characteristics of the program netlist. A complete description of this model and its generation can be found in [14].

A program netlist is a combinational circuit that compactly represents the software that is executed on the underlying hardware. In order to generate a program netlist, the control flow graph (CFG) is extracted from a low-level description of the software program, like assembly or machine code. Every node in the CFG represents an instruction of the program and the associated *program state* (PS). The PS includes the contents of data memory associated with the variables used in the program, and the *architecture state* (AS), defining the state of the processor's registers that are visible to the programmer. An edge between two CFG nodes indicates a possible execution from one instruction to another one. An additional Boolean signal called *active* is attached to PS in order to model the control flow of the program. This signal is propagated alongside the nodes in the program netlist and helps the SAT solver to efficiently explore the possible execution paths of the program. The *active* signal, when set to 1, indicates that a given node (instruction) belongs to the active execution path. In the case that a node has more than one successor (e.g., nodes related to jump/branch instructions), exactly one branch is active at any time.

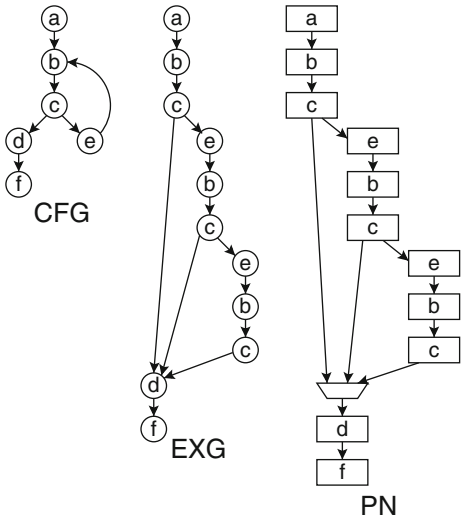


Fig. 5.1 Generating the program netlist (PN)

The CFG is fully unrolled into an *execution graph* (EXG). An EXG is a directed acyclic graph containing all possible execution paths of the program. An execution path always begins at a start state of the program and ends at an end state. The CFG is unrolled by unwinding the loops of the program. Figure 5.1 illustrates an example of unrolling. In order to reduce the complexity of the model, only branches that are part of at least one possible execution path are processed. A SAT solver can be used to identify such branches. Unrolling ends when all *active* branches have been processed and the end of the program has been reached. In addition, in order to minimise the size of the model, nodes belonging to identical program locations are merged. In this manner an EXG is obtained in which a single node may be shared by different execution paths. Merging is only allowed if it does not insert loops in the EXG.

A program netlist is then obtained from the EXG by replacing every node by its corresponding *instruction cell*. An instruction cell is a piece of combinational logic circuitry describing the functional behaviour of an ISA instruction according to the specific CPU architecture at hand. Consecutive instruction cells are connected by buses representing the program state.

A kind of instruction that is especially relevant to this work are load/store instructions which are used to communicate with the program's environment, e.g., the hardware periphery or other software layers. Instruction cells corresponding to such kind of instructions are equipped with additional input and output ports as shown in Fig. 5.2. These ports are called *pdata*, *ploc* and *pact* and represent respectively, the data value, the accessed location and the *active* flag indicating the activeness of the related instruction cell. Depending on whether the instruction cell reads or writes, *pdata* is an input or output signal. These three signals of an instruction cell constitute an *access port* for I/O memory locations.

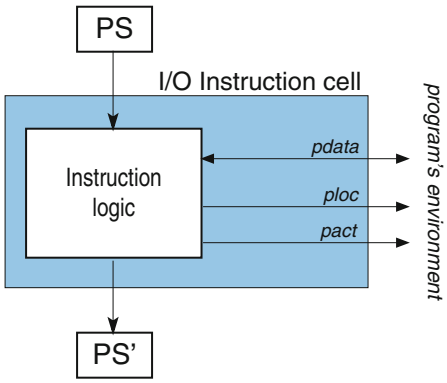


Fig. 5.2 Instruction cell with ports for accessing the environment

In the sequel, we use the term *program location* to indicate a memory location storing an instruction, and the term *memory location* to indicate an address corresponding to a location of the hardware periphery or a memory variable.

In the program netlist, instructions that access data memory require additional constraints so that the behaviour of the data memory is also modelled [18]. Therefore, for each instruction cell that reads from data memory there is a multiplexer structure that selects the last valid value written to the memory location being read by the instruction. In the case that a program depends on external events, e.g., by means of shared variables/channels, additional access ports of the respective instruction cell are left open or unconstrained as shown in Fig. 5.2. These access ports serve as the interfaces of the program, as will be further explained in the next section.

5.3 Software Property Language

This chapter presents how the interaction between hardware and software can be described in terms of I/O sequences and how the model of the sequences can be synthesised and combined with the underlying model of the software in order to perform formal property checking. As explained earlier, this is necessary in order to capture the reactive behaviour exhibited by hardware-dependent software. An additional advantage is that a model of the sequences allows to map the elements of the language to the elements of the underlying software model in a straightforward way. Subsequently, we show how a property language can be developed in terms of such sequences. The current working name for this language is RSPL (Reactive Software Property Language).

In the following, we introduce the main syntactic elements of RSPL. Since the programming language C is widely used for embedded software, our property language adopts many operators and syntax elements from C. For example, RSPL inherits from C the standard arithmetic, Boolean and comparison operators.

```
⟨Name of variable⟩'read
⟨Name of variable⟩'write
```

Fig. 5.3 Read/write attributes

5.3.1 Interfaces of a Hardware-Dependent Program

A property language for hardware-dependent software needs to provide a means for referring to the interfaces of a given program. In contrast to hardware description languages, software programs in high-level languages such as C do not explicitly capture their interfaces in a separate entity. For hardware-dependent software the elements of the interface correspond a set of addresses identifying, for example, registers inside hardware peripherals or shared memory locations used for communication with the operating system or with the application code. In view of the program netlist, such interface elements are modelled by means of access ports belonging to input/output instruction cells as explained in Sect. 5.2. In RSPL each of these addresses is assigned a name. In case of compiled machine code these names can be automatically obtained from the symbol table. Otherwise variable names can be defined manually by the user to enhance readability of the verification code. In order to distinguish the action of reading a variable (as input) from the action of writing a variable (as output), two variable attributes are introduced to the property language, namely *read* and *write*, as depicted in Fig. 5.3. This is the basis for referring to all input/output operations. Note that an address can be read and written several times. How this can be handled by the property language and how the verification engineer can refer to the different read and write instances is described below in Sect. 5.3.2.

There are also cases in which it is necessary to refer to the programmer-visible registers, for example when separately verifying a subroutine of a software driver. In such cases the content of a register can be expressed using the following syntax.

```
$⟨Name of register⟩'start
$⟨Name of register⟩'end
```

The attributes "*start*" and "*end*" indicate the start and the end of the program, respectively.

5.3.2 Sequences of Variables

The *sequence* is the key concept of our language; it is inspired by sequences in hardware property languages like SVA [15]. A sequence in SVA is constructed using the delay operator # which specifies the relative clock cycles (delay) between two events. However, we cannot directly import the semantics of sequences from SVA, since a sequence in SVA is defined over cycles which are relative to a global time reference such as a hardware clock. Models used for software (and in particular the program netlist) are not accurate with respect to hardware clock cycles, but

```
⟨Name of variable⟩'read #⟨n⟩
⟨Name of variable⟩'write #⟨n⟩
```

Fig. 5.4 Element accessor

rather instruction-accurate. Therefore, sequences are defined relative to the ordering of instruction executions. As illustrated in Fig. 5.4 we provide users with a way to define the individual elements of a sequence. Several such elements may be combined using Boolean operators in order to form sequences. We call the symbol # the *element accessor* for sequences and the natural number n represents the n-th element of a sequence. Since not every instruction accesses the interface of the program, the element n is the n-th occurrence of the associated interface variable along an execution path of a program (as opposed to the n-th instruction along that path). A software tool evaluating properties written in our language needs to map the elements of a sequence to the respective access ports in the program netlist. Because of the merging mechanism used to generate a program netlist, an input/output instruction cell can belong to several different execution paths. In other words, along an execution path an access port might be the i-th sequence element of a variable, whereas along another path, it may correspond to the j-th ($j \neq i$) sequence element of the same variable.

In the following we present an algorithm to map elements of sequences to the corresponding access ports in the program netlist. This algorithm is the basis for building the property logic for a SAT-based proof engine. To simplify the presentation, in the sequel, we only consider the memory locations of input/output variables, not their symbol names, since this relationship can be established easily through the symbol table. We use the term *memory location* to refer to both, an input or output variable as defined in Sect. 5.3.1, if the use is clear from the context.

The first step in the algorithm is a topological sorting of the nodes in the execution graph. We assign every node a unique index m, with $m \in \mathbb{N}$, so that along every execution path the (instruction) node indexed with i is executed earlier than the node indexed with j, if $i < j$. In the sequel, we refer to a node by its index in the topological order. Each memory location Loc_k is associated with a set of nodes accessing this location, i.e., $W = \{i_1, i_2, \ldots, i_{|W|}\}$ with $i_j < i_{j+1}$ and $1 \leq j < |W|$. The access port AP_{i_j} for a node i_j is composed of $pdata_{i_j}$, $pact_{i_j}$, and $ploc_{i_j} = Loc_k$, corresponding to the signal names in Fig. 5.2.

Given a memory location Loc_k, to map the n-th element ($1 \leq n \leq |W|$) of Loc_k's sequence to an access port AP_{i_j}, the challenging task is to find the index i_j of the node related to this element. The function *comp_index(n)*, depicted in pseudo-code notation in Algorithm 2, performs this task. It is generated for every memory location Loc_k. Function *port_mapping()* of Algorithm 3 is based on *comp_index()*. It connects the n-th element of the sequence to an access port in the program netlist.

The formulation of function *comp_index()* is based on the fact that at any time exactly one execution path of a program is active. An active path is characterised by the nodes in the program netlist whose *active* flags are asserted. In summary,

Algorithm 2 Compute index of the node associated with n-th sequence element

```
1: function COMP_INDEX(n)
2:     if n = 1 then
3:         if pact_{i_1} = true then
4:             return i_1
5:         else if pact_{i_2} = true then
6:             return i_2
7:             ...

8:         else if pact_{i_|W|} = true then
9:             return i_|W|
10:        else
11:            return 0
12:        end if
13:    else
14:        if pact_{i_n} = true ∧ compindex(n − 1) < i_n then
15:            return i_n
16:        else if pact_{i_n+1} = true ∧ compindex(n − 1) < i_n+1 then
17:            return i_n+1
18:            ...

19:        else if pact_{i_|W|} = true ∧ compindex(n − 1) < i_|W| then
20:            return i_|W|
21:        else
22:            return 0
23:        end if
24:    end if
25: end function
```

Algorithm 3 Map sequence to access port

```
1: function PORT_MAPPING(n)
2:     if compindex(n) = i_1 then
3:         return pdata_{i_1}
4:     else if compindex(n) = i_2 then
5:         return pdata_{i_2}
6:         ...

7:     else if compindex(n) = i_|W| then
8:         return pdata_{i_|W|}
9:     else
10:        return UNDEFINED
11:    end if
12: end function
```

comp_index() works as follows: To determine the n-th element of a sequence along any execution path, we first check the *active* flag of the node with index i_n, this is the very first node that could be the n-th element of a sequence. We also examine whether the $(n-1)$-th element along that path exists already, by checking whether the index of the node associated with $(n-1)$-th element is smaller than the index of the current node. If both conditions are met, then the n-th element of the sequence is

known to exist and the respective index can be returned. Otherwise we move on to the next candidate until a node related to the element we search is found or does not exist. With the *comp_index* function we can test whether an element of a sequence exists on a given path (by testing whether the result of *comp_index* is zero). The function is also used for verifying the execution order (cf. Sect. 5.3.3) of sequence elements that are related to different memory locations (variables).

With the ability of obtaining the index i_j of the node related to the n-th element of a sequence, it is straightforward to map the n-th element of the sequence to the access port AP_{i_j}. Function *port_mapping* depicted in Algorithm 3 performs this task.

In the remainder of this paper, for simplicity, we use the term *variable* for both, an element of an input/output sequence, or the state of a register at the start node/end node of the program.

5.3.3 Execution Order

Besides being able to relate software accesses to the same location at different points in time, in many cases it is also important to specify a temporal order of accesses to different memory locations. For instance, in order to issue a new transaction, a peripheral device may require that its driver first write the configuration/data register of the device at memory location Loc_1, and then set the start flag at memory location Loc_2; not maintaining this order could result in undefined behaviour of the device. Obviously, the property specification "**Loc_1'write#1 == Config_Data && Loc_2'write#1 == Start**" is insufficient for this requirement, since this statement does not define which of the two accesses, "**Loc_1'write#1**" and "**Loc_2'write#1**" is to be executed first by the software driver.

In RSPL, temporal ordering of accesses to different locations can be specified using the *execution order* section in a property. A user may specify an execution order between an input and an output, two inputs related to two different memory locations and two outputs related to two different memory locations. Checking the execution order is implemented by comparing the results of the functions *comp_index* for the respective sequence elements. Taking the example from above, if the returned value of the function *comp_index* for "**Loc_1'write#1**" is smaller than the returned value of this function for "**Loc_2'write#1**", then "**Loc_1'write#1**" is executed first. The syntax definition and an example of an *execution_order* specification are given in Fig. 5.5.

5.3.4 Safety and Liveness Properties

So far, we introduced the basic concepts and building blocks of the property language. In this section, we will present how to use them to build a property, and we will discuss what kinds of properties we can specify.

```
// execution order definition
execution_order:
⟨sequence element⟩ > ⟨sequence element⟩;

// examples
execution_order:
config'write#1 > Start'write#1;
config'write#2 > Start'write#1;
config'read#1 > config'write#1;
```

Fig. 5.5 Execution order

```
// Example of property

property example1;

// assumption part
assume:
$R5'start == 2;

// prove part
prove:
data_out'write#1 == data_in'read#1 + 4;
$R4'end == 0;

// execution order part
execution_order:
data_out'write#1 > data_in'read#1;
endproperty

// safety property
inst1: always example1;
// liveness property
inst2: eventually example1;
```

Fig. 5.6 Safety- /liveness-property

A property begins with the keyword *property*, followed by a valid identifier. The general structure of the property body follows an assumption/guarantee style. The body consists of two optional sections, *execution_order* and *assume*, and one mandatory section, *prove*. The *assume* part specifies the circumstances under which the assertion as specified in the *execution_order* and *prove* parts is to be checked. If we denote the assumption part by a predicate a, the prove part by c and the execution order part by o. Then a property p is translated to a Boolean formula $p := a \rightarrow (c \wedge o)$.

Given a property p, we can instruct the property checker to check it as a safety property or as a liveness property. As illustrated in Fig. 5.6, a safety property is indicated by the keyword "*always*", and a liveness property is indicated by the keyword "*eventually*" .

The semantic of "safety/liveness" is defined by evaluating the execution paths of a program. In contrast to Kripke models used in LTL or CTL model checking, the program netlist contains a finite number of paths of finite length. This greatly simplifies the evaluation of safety and liveness properties. A safety property "*always p*" means that on every execution path (from a start state to an end state of the program), the property p holds. This is similar to the LTL property $G\,p$, however, applied to a finite-length path. A liveness property "*eventually p*" means that there exists at least one execution path on which the property p holds. This is similar to the meaning of the CTL property EGp. It is straightforward to check the safety property using a SAT solver. In order to check a liveness property, we check the safety property "*always* $\neg p$". In case this property holds, we may conclude that the corresponding liveness property fails.

5.3.5 Syntax Extensions

With the language elements presented so far, we are able to capture the reactive behaviour of the software programs considered by our technique. We now present a number of extensions to the syntax that do not increase expressiveness but make property notation easier and more compact.

In the following, a variable *var* represents either an input (with attribute *read*) or an output (with attribute *write*). The symbol $\bowtie$ represents an arbitrary comparison operator, and *expr* represents any valid expression at either side of a comparison operator. The accessors depicted in Fig. 5.7 can be used to access a range of sequence elements related to a variable *var*. Every element is compared with the expression *expr*; if all comparison operations evaluate to true, then the result of this statement is true, otherwise it evaluates to false.

The dual case is handled by the accessor depicted in Fig. 5.8: it evaluates to true if the comparison operations return true at least N times in sequence.

```
var#[m:n] ⋈ expr :=
(var#m ⋈ expr) && (var#(m+1) ⋈ expr) &&
... && (var#n ⋈ expr)

with  m ≤ n and m,n ∈ ℕ
```

Fig. 5.7 Access a range of elements (universal)

```
var#EN[m:n] ⋈ expr :=

(var#m ⋈ expr) || (var#(m+1) ⋈ expr) ||
... || (var#n ⋈ expr)

with  m ≤ n, 0 < N < m − n, K ≥ N, and m,n,N ∈ ℕ,
where  K represents the number of terms evaluated true
```

Fig. 5.8 Access a range of elements (existential)

The function **exists** tests whether a sequence element exists on an execution path. In a safety property **exists** (**var#1**) checks whether **var#1** exists for every execution path, whereas in a liveness property it checks whether this element can be generated/consumed at least once during the execution of the program. This function can also be used to check whether the *assume* part of a property always evaluates to false due to non-existing sequence elements. Again, a SAT-based property checker can implement these checks using *comp_index()*.

5.4 Completeness of Property Sets

The completeness of a set of properties for a hardware design can be proven by using design-independent approaches such as [4, 6]. Our approach strongly relates to [4], which is explained in more detail in Sect. III-D of [17]. This method proves that two models of a design satisfying a set of properties $\{p_i\}$ are sequentially equivalent in terms of input/output sequences. What input and output values are considered and at what time points (clock cycles) is specified by the user in terms of so-called *determination conditions*. For example, a "data" signal needs to be uniquely determined by the design whenever a corresponding "valid" signal is asserted. A complete property set fulfils this determination condition if every property specifies the expected "data" value at the time points when the "valid" signal becomes asserted. Note that a property set can be checked for completeness independently of any design implementation for which this set of properties holds, because only relationships between signal names in the properties are checked. This idea can also be transferred to software properties written in RSPL and results in the following definition for completeness of a set of properties:

Definition 5.1. A set of RSPL properties is called complete, iff

1. there exists a property with a matching *assume* part for every possible input sequence applied to the program, and
2. every property uniquely specifies every output sequence produced by the program under the input sequences specified in the *assume* part.

Testing the two conditions of Definition 5.1 can be directly implemented in two checks called *Determination Test* and *Case Split Test*.

5.4.1 Determination Test

Unlike hardware that generates output sequences for every time point (clock cycle), the low-level software may produce output sequences of varying length, depending on the input sequences applied to the program. For instance, depending on configuration data given by the program's environment, a software driver may

perform burst write operations with 2 or 4 beats of data transfer, causing sequences with 2 or 4 elements, respectively. If we use design-independent methods, the completeness checker needs to know how many elements of sequences should every property at least specify. Denoting these values for every output in every property is tedious and error-prone. Therefore, for checking completeness of RSPL property sets, we give up on the design independence of a completeness criterion. Instead, we make use of the software model to determine how long the checked sequences are on each program execution.

In order to ensure that every output signal is uniquely determined by the property set, we perform two steps. First, we ensure that every property describes every element of all output sequences that are produced under all matching input sequences specified in the assumption part of the property. We solve this problem with the help of the design under verification, M, and the *comp_index* function. For simplicity, in the following we assume that the properties are written in a *causal* form, expressed as an implication between a cause (property assumption) and an effect (property commitment). This causal form is given if the input sequences are specified in the *assume* part of the property and the expected output behaviour is specified in the *prove* and *execution_order* parts. Given a property $p := a \rightarrow (c \wedge o)$, by syntactic analysis, we can identify the maximum sequence length k of any output sequence specified in the property. Then we check whether k is the maximum element generated by the software model M under the assumption a. For this purpose, we resort to the *comp_index(k+1)* function with respect to the assumption a: If this function returns a non-zero value, it means that the program can output a sequence with at least $k+1$ elements. The property under consideration does not specify this output sequence element, hence, we have detected a verification gap. Similarly, if a property does not at all mention some output produced by the program, we can detect this by checking for non-zero return of *comp_index(1)* for that output. A list of all possible outputs of a program can be easily obtained when synthesising the model of the program.

Once we have certified that every property describes every possible element of all output sequences, we check whether these output sequences are determined *uniquely*, i.e., whether, along any execution path, the property specifies exactly one value for every element of the output sequence. This can be done for every property independently of the software model. Let p be a property containing a set of signals $\{v_i\}$ (composed of a set of inputs $\{x_j : j \in \mathbb{N}, j \leq m\}$ and a set of outputs $\{o_n : n \in \mathbb{N}, n \leq l\}$) and corresponding sequence elements $\{v_i^{k_{v_i}} : k_{v_i} \in \mathbb{N}_{\neq 0}, k_{v_i} \leq t_{v_i}\}$. We create a copy p' of p by considering a copied set $\{v_i'\}$ of the variables appearing in the property and imposing the property on the copied variable set. The property p determines the outputs $\{o_n\}$ uniquely, iff the following formula is a tautology

$$(p \wedge p' \wedge \bigwedge_{j=0}^{m} \bigwedge_{k_{x_j}=1}^{t_{x_j}} (x_j^{k_{x_j}} = x_j'^{k_{x_j}})) \rightarrow \bigwedge_{n=0}^{l} \bigwedge_{k_{o_n}=1}^{t_{o_n}} (o_n^{k_{o_n}} = o_n'^{k_{o_n}})$$

5.4.2 Case Split Test

The case split test checks whether the property set covers every possible input sequence. Given a set of properties p_i with their respective assumption parts a_i, the case split test is conducted by proving that the formula $\bigvee_i a_i$ is a tautology.

5.4.3 Completeness Criterion

Theorem 5.1. *If and only if a set of RSPL properties $\{p_i\}$ passes both the Determination Test and the Case Split Test, then the property set is complete according to Definition 5.1.*

Proof. The theorem is true by Definition 5.1 because the Case Split Test checks for fulfilment of condition 1 of Definition 5.1 and the Determination Test checks for fulfilment of condition 2. □

5.5 Case Study

The property language developed in this work has been successfully applied to specifying properties for an industrial software driver for a *LIN* master node. The software was developed by Infineon Technology AG. Note that the focus of the work is on the challenges of specifying complete sets of properties for this type of software, not on the proving techniques. The properties shown in this case study have already been proven earlier [14], based on a manual construction of checker automata which were added to a program netlist model of the software. In this work we present the formulation of these properties in RSPL.

The hardware peripheral controlled by the software driver is a UART (Universal Asynchronous Receiver/Transmitter), connected to the physical LIN bus lines. A LIN bus is composed of one master node and several slave nodes. Data is transmitted on the LIN bus in so-called *frames*. A frame is composed of several fields: a header, up to 8 bytes of data, and a checksum. The master node is responsible for sending the header field which is composed of a *break* field indicating the start of new frame, a *sync byte* field used for synchronisation, and an *identifier* (ID) field. Slave nodes evaluate the identifier field and, if there is a match, then the corresponding slave node either sends or receives data. The LIN driver code under consideration implements a master node. It supports six fixed-valued IDs. It can send or receive 2, 4 or 8 bytes of data for each of the six IDs. Data is communicated with the application software through shared memory locations that serve as the interface of the LIN driver.

We now consider a first property in Fig. 5.9 specifying the transmission of a frame, according to the protocol specification for the LIN bus. For reasons of space,

```
property lin_master_transmits_2_bytes;

assume:
s_id'read#1 == C_ID0;

prove:
// master task
uart'write#1 == C_BREAK;
uart'write#2 == C_SYNC;
uart'write#3 == C_ID0;

// slave task

// an example for undetermined uart'write#4 would be
// uart'write#4 == 1 || uart'write#4 == 0;
uart'write#4 == data1'read#1;
uart'write#5 == data2'read#1;
uart'write#6 == CHECKSUM;

execution_order:
data1'read#1 > uart'write#4;
data2'read#1 > uart'write#5;
endproperty;

// safety property
inst1: always lin_master_transmits_2_bytes;
```

Fig. 5.9 LIN_TX_Frame_2_Bytes

we show only the case for data length of 2 bytes. Furthermore, for readability, we use the names of variables and constants instead of their memory addresses. Variables *data1* and *data2* store the payload data provided by the application software. The *s_id* are shared variables storing the ID that needs to be transferred to the slave task. The symbol *uart* refers to the Tx/Rx buffer of the UART.

In the following, the prefix "C_" indicates a constant value. *C_ID0* identifies a 2-byte transmission. *CHECKSUM* abstracts the "checksum" computation.

We also need to define an *execution_order* section in the property in order to specify that the data must be available before it is transmitted.

Note that a program that does not support *C_ID0* at all may nevertheless fulfil the property in Fig. 5.9. We therefore need to check the liveness property in Fig. 5.10 in order to make sure that at some point in time a *C_ID* = *frame* is indeed sent to the UART.

Figure 5.11 shows the use of the *exists* function in a property that checks whether the driver is capable of transmitting 8-byte data frames.

Obviously, the safety property in Fig. 5.9 does not completely specify the entire program. It specifies only the case that the LIN master transmits 2 bytes of data to a slave. The case split test presented in Sect. 5.4 identifies the missing cases by checking whether there exists a corresponding property for every value of *s_id*.

```
property lin_test_C_ID0;

prove:
uart'write#3 == C_ID0;
endproperty;

// liveness property
inst2: eventually lin_test_C_ID0;
```

Fig. 5.10 LIN liveness

```
property lin_test_support_8_bytes;

prove:
exists(uart'write#11);
endproperty;

inst3: eventually lin_test_support_8_bytes;
```

Fig. 5.11 LIN liveness 2

The comments section of this property shows an example of a failing determination test for a sequence element, where this statement states that the value of "**uart'write#4**" could be either 0 or 1. Thus, the value of this variable is not *unique*: Suppose that "**uart'write#4**" is a Boolean variable, then the expression in the statement evaluates to *true*. Hence, the property proves nothing about this variable.

5.6 Conclusion

In this chapter we presented the concept and the basic framework of a software property language for reactive low-level embedded software. The language is based on a computational model for this type of software, called *program netlist*. The language allows to easily express the I/O behaviour of the software by using temporal sequences. Furthermore, properties can be synthesised and combined with the program netlist into a single model allowing to perform formal verification. Taking advantage of the temporal description we defined a completeness criterion for a set of properties. We showed how to use the elements of the language to formally and completely specify a LIN master node.

The future development of the proposed property language will include extensions in order to support compositional verification for cases where the overall verification of a program needs to be partitioned to improve scalability. Additionally, the concept of *functions* or *macros* will be introduced for structuring and re-using verification code.

References

1. Accellera Organization Inc. Property Specification Language - Reference Manual, Version 1.1. http://www.eda.org/vfv/docs/PSL-v1.1.pdf, June 2004
2. Ball, T., Rajamani, S.K.: Slic: A specification language for interface checking (of c). Technical Report MSR-TR-2001-21, Microsoft Research, Jan (2002)
3. Beyer, D., Chlipala, A., Henzinger, T., Jhala, R., Majumdar, R.: The blast query language for software verification. In: Giacobazzi, R. (ed.) Static Analysis. Lecture Notes in Computer Science, vol. 3148, pp. 2–18. Springer, Berlin (2004)
4. Bormann, J., Busch, H.: Verfahren zur Bestimmung der Güte einer Menge von Eigenschaften (Method for determining the quality of a set of properties). European Patent Application, Publication Number EP1764715, 09 (2005)
5. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Commun. ACM **56**(2), 82–90 (2013)
6. Claessen, K.: A coverage analysis for safety property lists. In: Proceedings of International Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 139–145. IEEE Computer Society, Washington, DC, (2007)
7. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ansi c programs. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 168–176. Springer, Berlin (2004)
8. Clarke, E.M., Emerson, E.: Synthesis of synchronization skeletons for branching time temporal logic. In: Logics of Programs. Lecture Notes in Computer Science, vol. 131. Springer, Berlin (1981)
9. Haedicke, F., Große, D., Drechsler, R.: A guiding coverage metric for formal verification. In: DATE, pp. 617–622, 2012
10. Holzmann, G.J.: The SPIN model checker. IEEE Trans. Softw. Eng. **23**, 279–295 (1997)
11. Katz, S., Grumberg, O., Geist, D.: "have i written enough properties?" - a method of comparison between specification and implementation. In: Proceedings of Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME), pp. 280–297. Springer, London (1999)
12. Kurshan, R.P.: Computer-Aided Verification of Coordinating Processes – The Automata-Theoretic Approach. Princeton University Press, Princeton, NJ (1994)
13. Schlich, B.: Model checking of software for microcontrollers. ACM Trans. Embed. Comput. Syst. **9**(4), 36:1–36:27 (2010)
14. Schmidt, B., Villarraga, C., Fehmel, T., Bormann, J., Wedler, M., Nguyen, M., Stoffel, D., Kunz, W.: A new formal verification approach for hardware-dependent embedded system software. IPSJ Trans. Syst. LSI Des. Methodol. **6**, 135–145 (2013)
15. Spear, C.: SystemVerilog for Verification: A Guide to Learning the Testbench Language Features. Springer, Berlin (2008)
16. The MathWorks, Inc. USA: Polyspace - Static Analysis Tools (2014). http://www.mathworks.com/products/polyspace/ (2014)
17. Urdahl, J., Stoffel, D., Kunz, W.: Path predicate abstraction for sound system-level models of rt-level circuit designs. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **33**(2), 291–304 (2014)
18. Villarraga, C., Schmidt, B., Bartsch, C., Bormann, J., Stoffel, D., Kunz, W.: An equivalence checker for hardware-dependent software. In: 11. ACM-IEEE International Conference on Formal Methods and Models for Codesign, pp. 119–128 (2013)
19. Yang, F.Z., Ganai, M., Gupta, A., Shlyakhter, I., Ashar, P.: FSoft software verification platform. In: Proceedings of International Conference Computer Aided Verification (CAV), pp. 301–306. Springer, Berlin (2005)

Chapter 6
Exploiting Electronic Design Automation for Checking Legal Regulations: A Vision

Oliver Keszocze and Robert Wille

Abstract Legal regulations are large and complex documents that require experts such as lawyers to be understood. Working with these documents is a manual and time- consuming task. Common use cases are to decide whether a submission is conform with the regulations or to check whether certain corner cases are possible in the given set of rules. We envision to address many of these problems by treating legal regulations in the same manner as system specifications. This allows to apply sophisticated formal methods from *Electronic Design Automation* (EDA). For this, we briefly discuss the process of (semi)-automatically formalizing legal regulations. Afterwards, we illustrate the correspondence of various problems in the considered domain (here: regulations on scales and fees for medical doctors) with well-known EDA problems. We sketch the application of formal methods by means of examples and envision that in the future, the exploitation of formal methods to analyse legal regulations will greatly help lawmakers and "end users" alike.

6.1 Introduction

In *Electronic Design Automation* (EDA), circuits and systems are realised based on an initially given specification. During the design process, this specification is implemented using proper hardware or system description languages such as SystemC [10], System Verilog [11], or VHDL [9]. Due to the strive for higher levels of abstraction, the application of modelling languages such as the *Unified Modeling Language* (UML, [15]) or the *Systems Modeling Language* (SysML, [19]) found recent interest in the design of circuits and systems.

O. Keszocze (✉)
Group for Computer Architecture, University of Bremen, Germany

Cyber-Physical Systems, DFKI GmbH Bremen, Germany
e-mail: keszocze@informatik.uni-bremen.de

R. Wille
Institute for Integrated Circuits, Johannes Kepler University Linz, Austria

Cyber-Physical Systems, DFKI GmbH Bremen, Germany
e-mail: robert.wille@jku.at

© Springer International Publishing Switzerland 2016

F. Oppenheimer, J.L. Medina Pasaje (eds.), *Languages, Design Methods, and Tools for Electronic System Design*, Lecture Notes in Electrical Engineering 361,
DOI 10.1007/978-3-319-24457-0_6

All these developments have in common that they transform a specification, which is usually provided in a natural language, into a formal representation. At the same time, the resulting formal descriptions are subject to various correctness checks for which a variety of (automatic) EDA methods have been proposed in the past. More precisely, very powerful methods are in practical use today which check, e.g. the equivalence of two realizations (see, e.g., [2]), prove whether certain transitions in a system are possible (see, e.g., [1]), or generate test patterns that identify faults in the physical realization (see, e.g., [7]). These EDA methods became an integral part of today's design flow and emerged to powerful tools which can handle designs of considerable complexity.

At the same time, these methods also provide significant potential to application areas beyond EDA. In this chapter, we investigate such an application area, namely the consideration of legal regulations.

Legal regulations can be understood in a similar fashion as a specification for a technical system. The only difference is that legal regulations describe rules and requirements of a certain aspect of the daily life instead of something technical such as a circuit or system. But this difference is just of superficial relevance for the applied EDA methods. In fact, also legal regulations are initially provided in natural language which can be formalized (first ideas have been already proposed, see, e.g., [12, 17]). Based on this formal representation, checks for "correctness", e.g. plausibility, uniqueness, the existence of desired scenarios, and many more can be conducted.

In this chapter, we envision and discuss possible exploitations of existing EDA methods in order to check legal regulations. For this purpose, we briefly review a selection of EDA methods and sketch available approaches aiming for the formalization of legal regulations. Based on that, we discuss and illustrate how the EDA methods available for the design of circuits and systems together with a formal representation of legal regulations can be exploited. All considerations are conducted using the German Regulations on Scales of Fees for Medical Doctors (German: Gebührenordnung für Ärzte [8]) as an example.

The remainder of this chapter is organized as follows. In Sect. 6.2, we briefly review EDA methods that are used to verify the correctness and behaviour of circuits and systems. The reviewed solutions are based on Boolean satisfiability. Afterwards, in Sect. 6.3, the domain of interest, legal regulations that control how medical doctors may invoice their services, is introduced. Furthermore, the method of extracting a formal model from natural language texts is sketched. Then, Sect. 6.4 introduces a variety of applications for EDA methods in this context and explains them by means of examples. Finally, in Sect. 6.5, the work is concluded.

6.2 Typical Methods for Electronic Design Automation

Every year in the past decades, the design of circuits and systems grew more and more complex. The resulting complexity eventually motivated the development of methods for *Electronic Design Automation* (EDA). Moreover, search spaces

to be traversed became larger and larger so that, today, methods are in place which employ dedicated search and learning strategies in order to cope with the underlying computational complexity. Amongst many other methods, solvers for *Boolean satisfiability* (so called *SAT solvers*; see, e.g., [5, 6]) represent one of the most popular of these methods. In this section, we briefly review the basics on Boolean satisfiability, their solving schemes, as well as application areas. This provides representatives of EDA methods whose application to the domain of legal regulations is envisioned afterwards.

6.2.1 Boolean Satisfiability and SAT Solvers

The *Boolean satisfiability problem* (SAT problem) is to determine an assignment α to the variables of a Boolean function f such that $f(\alpha)$ evaluates to true or to prove that no such assignment exists. Often, f is given in *Conjunctive Normal Form* (CNF). A CNF consists of a conjunction of clauses. A clause is a disjunction of literals and each literal is a propositional variable or its negation.

Example 1. Let $f = (x_1 \vee \bar{x}_2 \vee x_3)(\bar{x}_1 \vee x_2)(\bar{x}_2 \vee \bar{x}_3)$. Then $x_1 = 1 = x_2 = 1, x_3 = 0$ is a satisfying assignment for f. The value of x_1 ensures that the first clause becomes 1 while x_2 ensures this for the second as does x_3 for the third clause.

The SAT problem is one of the central NP-complete problems. In fact, it was the first known NP-complete problem as was proven by Cook in 1971 [3]. Despite this proven complexity, SAT algorithms are nowadays capable of handling practical problem instances of considerable size, i.e. SAT instances which are composed of hundreds of thousands of variables, millions of clauses, and tens of millions of literals. Most of these SAT solvers are based on backtracking algorithms and use three essential procedures: (1) the decision heuristic assigns values to free variables, (2) the propagation procedure determines implications resulting from the last assignment(s), and (3) the conflict analysis tries to resolve conflicts that occur during the search by clever backtracking schemes. Advanced techniques such as *efficient Boolean constraint propagation* [14] and *conflict analysis* [13] are common in state-of-the-art SAT solvers (see, e.g., [5, 6]) and strongly contribute to their effectiveness.

6.2.2 Applications in EDA

The efficiency as well as the clever traversal schemes of SAT solvers have found numerous application for many computationally complex problems—including several EDA problems. In the following, just a selection is briefly sketched:

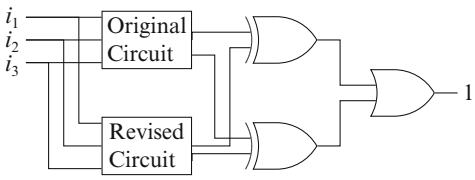


Fig. 6.1 Miter structure

- *Equivalence Checking*
  During the design process, the originally determined circuit is usually revised a couple of times (e.g. for the purpose of optimization). After each revision, it is important to prove whether or not the revised circuit is still functionally equivalent to the original one. This problem can be addressed by formulating the resulting equivalence checking problem by means of a so-called *miter structure* [2]. Here, it is made sure that always the same input assignments are applied to both circuits. Furthermore, XORs are added to corresponding output pairs in order to detect possible differences. Figure 6.1 shows the resulting structure. Then, if at least one XOR evaluates to 1 (determined by an additional OR operation), the two circuits are not equivalent. As this obviously represents a satisfiability problem ("Does there exist an assignment to the inputs of the circuits such that the output of the miter structure evaluates to true?"), the resulting SAT instance can easily be formulated. Then, all possible inputs assignments are *symbolically* considered which guarantees that a possible assignment leading to different outputs is determined when it exists.
- *Reachability Analysis*
  Recent developments of formally specifying complete systems using description languages allow for further automated analysis. One particular problem is to check whether certain states of the system, desired or undesired, are reachable [1]. The procedure is to symbolically translate all possible transitions of the system into a SAT formulation. Afterwards, a clause representing the state of interest is added. If the whole SAT formulation is satisfiable, the given state is reachable. In case of undesired behaviour, this indicates a bug in the system.
- *Automatic Test Pattern Generation (ATPG)*
  As a final example, ATPG considers the determination of an assignment to all primary inputs of a given circuit which, based on a given fault model, will show a possible erroneous behaviour at the primary outputs of this circuit [7]. This problem can also be formulated as a corresponding SAT problem ("Does an assignment showing the erroneous behaviour exist?"). ATPG is an interesting example, since existing EDA methods for this problem are capable of handling circuits and systems composed of several hundreds of thousands of components.

6.3 Formal Representation of Legal Regulations

Usually, legal regulations are provided in natural language. But in order to exploit EDA methods for checking them, a formal representation is required. Recently, researchers investigated several strategies and schemes how to efficiently formalize legal regulations (see, e.g., [12, 17]). Besides that, many software tools exist in which legal regulations (e.g. tax code) have (manually) been incorporated in order to support "end users" (e.g. citizens preparing their tax returns). In this section, we discuss existing formalizations of legal regulations which, eventually, provide possible inputs for EDA-inspired verification methods. We exemplary focus on legal regulations from the domain of doctoral fees, namely the *German Regulations on Scales of Fees for Medical Doctors* (German: Gebührenordnung für Ärzte [8], abbreviated as GOÄ in the following).

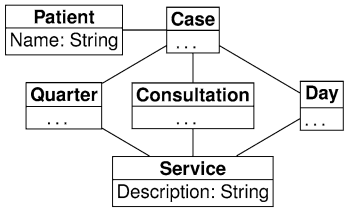
The GOÄ specifies how doctors are supposed to generate invoices for their services. The GOÄ eventually defines which services can be accounted and to what extent. As an example, Fig. 6.2a shows a particular regulation from the GOÄ.[1] Medical doctors, hospitals, etc., are supposed to prepare their invoices according to these regulations.

However, before the submitted invoices are indeed paid, corresponding authorities check whether the resulting invoices are in line with the regulations of the GOÄ. Due to the large amount of regulations, this is usually performed automatically

a

"The service 45678 is not to be billed together with GOÄ position 54687 within the third and fourth quarter of 2014." (*Die Leistung 45678 ist im dritten und vierten Quartal 2014 nicht neben Position 54687 berechnungsfähig*)

b



c

```
if (
      Case.quarter >= Quarter(2014,3) &&
      Case.quarter <= Quarter(2014,4)
) {
      if (together(45678,54687) {
            // do not bill and report error
      }
}
```

Fig. 6.2 Considered domain and problem. (**a**) A GOÄ rule describing a mutual exclusion of services (German sentence in parentheses). (**b**) Simplified domain of the GOÄ. (**c**) Resulting DSL expression

[1]Note that all regulations in the GOÄ are originally provided in German. However, in order to describe the proposed ideas, all examples have been translated.

by software tools. To this end, a formal representation of the respective GOÄ regulations has to be available.

Thus far, this formal representation is created manually—using a *Domain-Specific Language* (DSL) as well as a proper model which represents the structure of the respective processes (i.e. the performed services and the instances in time in which these services have been provided.[2]) Fig. 6.2b provides a sketch of the applied model: All possible *services* that doctors may perform are grouped by different criteria (e.g. *consultations, days*, or *quarters*) and collected in a single *case*. Each case is then related to a *patient*. A single service may require multiple consultations and a doctor may perform multiple services in a single consultation.

With this background, the original regulation from Fig. 6.2a can formally be represented as shown in Fig. 6.2b. Note that this expression is not uniquely determined and may look completely different, depending on the software developer's preferences. With formalizations like that, software tools checking invoices can easily be developed.

While this approach still requires significant manual effort in order to create the respective formal descriptions, another approach formalizing the corresponding GOÄ regulations has been proposed in [12]. Here, a (semi-) automatic method is proposed which is composed of two steps. In the first step, the sentences are preprocessed and grouped, while the second step uses methods from *Natural Language Processing* (NLP, [4, 16]) to create the DSL expression.

In the first preprocessing step, synonyms are normalized. Given a knowledge base (filled e.g. by an expert from the domain), words like "service" and "GOÄ position" can be matched in order to simplify the analysis. Afterwards, the sentences are simplified by shortening long enumerations. These enumerations do not carry any information in the NLP step besides "an amount of services". Therefore, terms such as "services 12345, 23456, 34567, and 98765" are reduced to placeholder-terms, e.g. by hash-tag-like identifiers such as "01234567.0". These are still understood as a number by NLP tools while not resulting in a complicated structure that renders the further analysis impossible. Further simplifications such as removing long descriptions of equipment are conducted as well.

After these preprocessing steps, actual natural language processing of the regulations (still provided in natural language) are performed. Here, established techniques from the domain of natural language processing such as *typed dependencies* [4] are employed. Typed dependencies create a relationship between words of a sentence and, furthermore, type them with respect to their grammatical relation. For example, the words "the" and "service" have a *determiner* dependency (short: *det*) with each other. All typed dependencies of a sentence eventually form a graph based on which a translation into a formal representation can automatically be conducted

[2]Note that further issues such as particular doctors and their qualification as well as special equipment are not considered in the following.

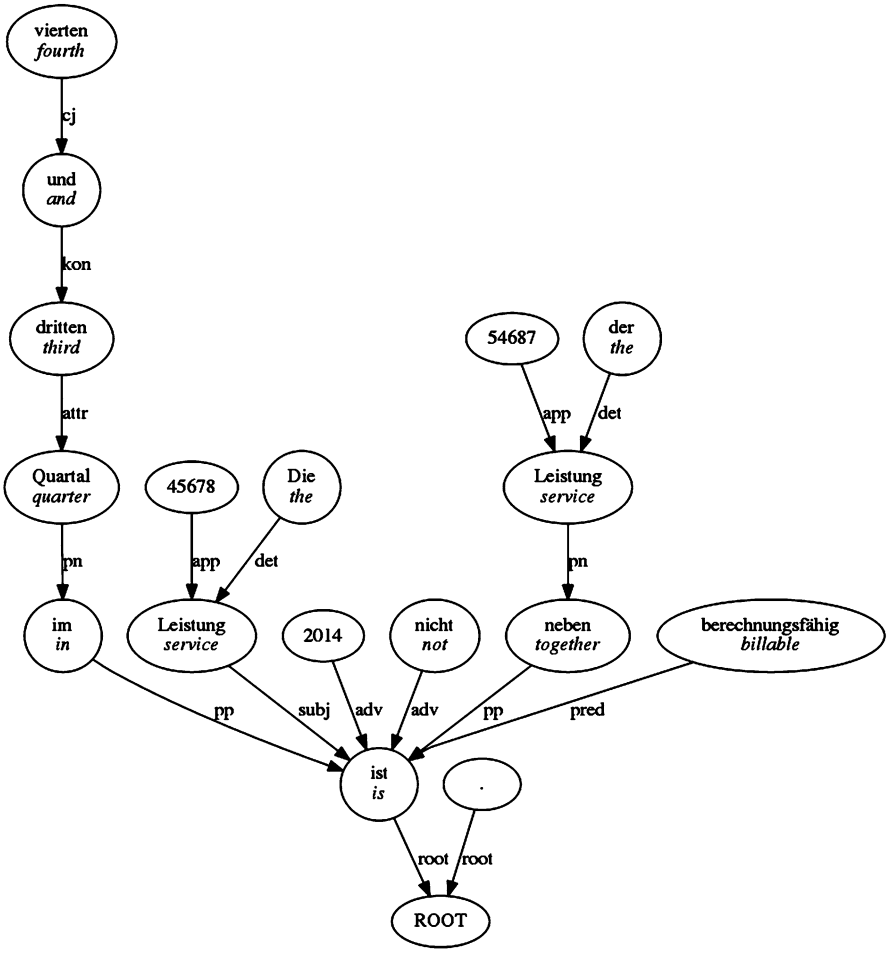


Fig. 6.3 Typed dependencies of the sentence shown in Fig. 6.2a

in many cases. The general idea is illustrated by means of the GOÄ rule of Fig. 6.2a. The corresponding typed dependency graph of the original German sentence, obtained using the tool ParZu [16], is shown in Fig. 6.3.

This analysis provides, e.g., the following information:

- The main reference, i.e. which service is of interest, is determined. This can be obtained by analysing the subject *subj* of the sentence.
- Other references to services might be found by following *pn-app* paths in the graph.
- A negation of the sentence is indicated by an *adv* path from the root note of the sentence to the word “not”.

```
context Patient inv:

-- fetch relevant quarters with invoices
let quarters =
       case.quarter.select(q| q.year=2014 and (q.n=3 or q.n=4))
in
  -- all services performed in these quarters
  let services = quarters.collect(service)
  in
    -- at most one service performed
    services.select(s | s.id=45678 or s.id=54687).size() <= 1
```

Fig. 6.4 Automatically generated formal notation of the sentence from Fig. 6.2a

Eventually, this enables an *automatic* derivations of a formal representation from a given legal regulation. In case of the considered sentence, a formal notation as sketched in Fig. 6.4 is created using OCL [18]. Note that other means of formal description are possible as well. Using the techniques described in [12], approximately 60 % of the regulations given in the GOÄ can automatically be formalized this way. The quality of the results heavily depends on the initial knowledge base used for determining synonyms and simplifying sentences.

6.4 Applying EDA Methods to the Formal Representation

As reviewed in the previous section, formal representations of legal regulations already exist and are applied, e.g. in order to check the validity of invoices. Similar application areas exist, e.g. in the domain of tax returns, applications to federal agencies in general, etc. However, all corresponding software tools basically check certain instances of invoices, tax returns, applications, etc., only. Exploiting the EDA methods reviewed in Sect. 6.2, much more potential exist. In fact, as for the design of circuits and systems, there is no guarantee that legal regulations are free of contradictions by themselves. The following example illustrates a possible problem:

Example 2. Consider the following four (simplified) GOÄ regulations:

- "Service *A* is not to be billed together with service *C* while, at the same time, it forces billing the service *B*".
- "Service *B* forces billing the service *C*".
- "Service *C* forces billing the service *D*".
- "Service *D* forces billing the service *A*".

Table 6.1a summarizes these regulations in a more compact fashion. The first column refers to the respective services, while the remaining columns provide the services which are mutually exclusive to them and the services which are forced to be billed in addition to them.

Table 6.1 Exemplary constraints for services

(a) Inconsistent model		
Service	Excludes	Forces
A	C	B
B	–	C
C	–	D
D	–	A
(b) Erroneous model		
Service	Excludes	Forces
A	C	B
B	–	C
C	A,B	–
D	C	–
(c) Example for invoice optimization		
Service	Costs $	Excludes
A	40	–
B	50	C
C	20	–
D	80	A,B

Obviously, this would be a set of contradictory legal regulations, since

- the billing of any service (*A,B,C*, or *D*) eventually forces the presence of all other services while,
- at the same time, service *A* forbids the existence of service *C*

While it is simple to detect contradictions for simple regulations as shown here, it surely becomes a crucial tasks when dozens or even hundreds of such regulations (usually written in hard language) have to be considered.

Using EDA methods, contradictions even in large (formalized) legal regulations could easily be detected. In fact, these methods do not care whether the corresponding formal representation has been derived from a hardware specification or a legal regulation. For them, legal regulations can be seen as a "specification" of how certain (real world) processes may be implemented in practice. As for circuits and systems, the goal of the methods is to detect the existence of contradictions or unwanted descriptions. While, in the domain of legal regulations, contradictions can not always be prevented (due to case-by-case decisions, trade-offs between certain rights, etc.), in particular in fields such as billings, tax returns, etc., a high degree on uniqueness is desired. In these cases, EDA methods may provide the basis for helpful tools to detect contradictions as discussed in Example 2. More precisely:

Example 3. Consider again the four GOÄ regulations from Example 2 and assume that corresponding formal representations (denoted by Booleans A, B, C, D) have been derived from them. Then, EDA methods allow to symbolically consider all possible applications scenarios and, by this, can proof, e.g. whether this set of

regulations is free of contradictions. In other words, the SAT problem "Does there exist at least one scenario which satisfies all regulations?" can be formulated. In a simplified form, this translates into the formulation

$$(A \implies \overline{C} \wedge B) \wedge (B \implies C) \wedge (C \implies D) \wedge (D \implies A)$$

for which SAT solvers can prove that no satisfying solutions exist. That is, not all regulations can be satisfied at the same time.

In a similar fashion, many other questions concerning legal regulations can be addressed by EDA methods. For example:

- *Checking for desired/undesired interpretations* of the given regulations. As an example, consider the regulations for services *A, B*, and *C* as shown in Table 6.1b. The intention of these regulations is to have two services excluding each other while services *A* and *B* force the billing of certain other services. Note that service *B* forces to also put service *C* on the invoice. Unfortunately, service *C* cannot be billed together with the other two services. This basically makes it impossible for service *B* to be ever billed. A formal model applied to an EDA method sketched above could find this inconsistency and present it to the corresponding authority. The authority then would probably decide that the intention was to have service *B* be mutually exclusive with service *C* and, therefore, move the *C* entry from the "Forces" to the "Excludes" column.

  This application is similar to the *reachability analysis* as reviewed in Sect. 6.2.
- *Fixing of certain properties.* Legal regulations are usually updated on a regular basis. Consider an update that introduces the constraint that service *A* must not be invoiced together with service *B*. When introducing this change (which, in real world documents, will not be as easily describable as this artificial example), one wants to ensure that it does not change any other rules. This means that no invoice that was legal before the update becomes invalid afterwards (while, at the same time, preserving the intended change). For checking this, a miter-like structure (see Sect. 6.2) can be employed. In this application, the "outputs" of interest are the valid invoices.
- *Automated optimization of invoices*, i.e. determine an application of regulations which leads to the best possible outcome for the doctor. As an example, consider the constraints as shown in Table 6.1c (which additionally provides the costs of a particular service). In this simple example, one can easily validate by hand that billing service *C*, and *D* yields the biggest outcome. For larger sets of regulations, this becomes an infeasible task for humans. A variant of this problem is to find the maximum amount of money that can be invoiced while staying within a certain budget.
- *Automated checks* whether a given invoice is consistent with the regulations. This helps the doctor as well as the officials deciding on the invoice at hand.
- *The automated generation of DSL expressions* (such as shown in Fig. 6.2c), which are used in software to check documents for validity. This does not only significantly reduce the development costs of software applications, but it

also ensures that the expressions are correct by eliminating the possibility of introducing errors in the creation/programming.

6.5 Conclusion

Formal descriptions of legal texts greatly improve the daily work for experts. This ranges from the direct end user of these regulations (such as a doctor as seen in the examples) to lawyers creating such texts. Applicants have the means to check whether their documents are valid before handing them in, thereby saving a lot of work on both sides. Decision makers such as politicians now have further means that aid them in understanding the full impact of their regulations. This is of great importance as it is difficult to completely anticipate the consequences of (parts of) regulations even for experts such as lawyers. We envision that in the near future, for most legal texts, formal descriptions will be available or work on formalizing them is in progress.

Acknowledgements The authors would like to thank Betina Keiner, Matthias Richter, Lucjan Suchy, and Gottfried Antpöhler for the many fruitful discussions. This work was supported by the German Federal Ministry for Economic Affairs and Energy (BMWi) under grant no. KF2054902MS2 and KF2013014MS2 as well as the German Research Foundation (DFG) under grant no. WI 3401/5-1.

References

1. Abdulla, P.A., Bjesse, P., Eén, N.: Symbolic reachability analysis based on SAT solvers. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 411–425. Springer, Berlin (2000)
2. Brand, D.: Verification of large synthesized designs. In: International Conference on CAD, pp. 534–537 (1993)
3. Cook, S.A.: The complexity of theorem-proving procedures. In: Symposium on Theory of Computing, pp. 151–158 (1971)
4. de Marneffe, M.C., MacCartney, B., Manning, C.D.: Generating typed dependency parses from phrase structure parses. In: Conference on Language Resources and Evaluation, pp. 449–454 (2006)
5. de Moura, L.M., Bjørner, N.: Z3: An Efficient SMT Solver. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 337–340 (2008)
6. Eén, N., Sörensson, N.: An Extensible SAT-solver. In: SAT 2003, LNCS, vol. 2919, pp. 502–518 (2004)
7. Eggersglüß, S., Drechsler, R.: Efficient data structures and methodologies for SAT-based ATPG providing high fault coverage in industrial application. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **30**(9), 1411–1415 (2011)
8. Gebührenordnung für Ärzte (GOÄ). Online available at http://www.e-bis.de/ (2014)
9. IEEE Standard VHDL Language Reference Manual Amendment 1: Procedural Language Application Interface (2007)
10. IEEE Standard for Standard SystemC Language Reference Manual (2012)

11. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language (2013)
12. Keszocze, O., Keiner, B., Richter, M., Antpöhler, G., Wille, R.: (Semi-)automatic translation of legal regulations to formal representations: expanding the horizon of EDA applications. In: Forum on Specification & Design Languages (FDL) (2014)
13. Marques-Silva, J.P., Sakallah, K.A.: GRASP: A search algorithm for propositional satisfiability. IEEE Trans. Comput. 48(5), 506–521 (1999)
14. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Design Automation Conference, pp. 530–535 (2001)
15. Rumbaugh, J., Jacobson, I., Booch, G. (eds.): The Unified Modeling Language Reference Manual. Addison-Wesley Longman, Essex (1999)
16. Sennrich, R., Schneider, G., Volk, M., Warin, M.: A new hybrid dependency parser for German. In: Proceedings of the German Society for Computational Linguistics and Language Technology, pp. 115–124 (2009)
17. Soltana, G., Fourneret, E., Adedjouma, M., Sabetzadeh, M., Briand, L.: Using UML for modeling procedural legal rules: approach and a study of Luxembourg's tax law. In: Model-Driven Engineering Languages and Systems, pp. 450–466. Springer, Berlin (2014)
18. Warmer, J., Kleppe, A.: The Object Constraint Language: Precise Modeling with UML. Addison Wesley, New York (1999)
19. Weilkiens, T.: Systems Engineering with SysML/UML: Modeling, Analysis, Design. Morgan Kaufmann, San Francisco (2007)

Part III
Parallel Architectures

Chapter 7
Synthesizing Code for GPGPUs from Abstract Formal Models

Gabriel Hjort Blindell, Christian Menne, and Ingo Sander

Abstract Today multiple frameworks exist for elevating the task of writing programs for GPGPUs, which are massively data-parallel execution platforms. These are needed as writing correct and high-performing applications for GPGPUs is notoriously difficult due to the intricacies of the underlying architecture. However, the existing frameworks lack a formal foundation that makes them difficult to use together with formal verification, testing, and design space exploration. We present in this chapter a novel software synthesis tool—called f2cc—which is capable of generating efficient GPGPU code from abstract formal models based on the synchronous model of computation. These models can be built using high-level modeling methodologies that hide low-level architecture details from the developer. The correctness of the tool has been experimentally validated on models derived from two applications. The experiments also demonstrate that the synthesized GPGPU code yielded a 28× speedup when executed on a graphics card with 96 cores and compared against a sequential version that uses only the CPU.

7.1 Introduction

We are experiencing a seemingly never-ending improvement in computational processing capacity. The past decades have yielded faster, denser, and more complex chips, and the processing units are increasingly being composed into multi-core platforms which require complicated communication and scheduling schemes. This results in an incredible challenge that system developers need to face in managing the growing complexity of systems. To make matters worse, low-level implementation details must be considered in order to produce, not only correct, but also efficient systems. This problem is especially notorious for *general purpose graphics processing units* (GPGPUs). GPGPUs are massively parallel execution platforms that have emerged from the graphics card technology whose processing capacity

G. Hjort Blindell (✉) • C. Menne • I. Sander
Department of Electronic Systems, School of Information and Communication Technology, KTH Royal Institute of Technology, Stockholm, Sweden
e-mail: ghb@kth.se; chris.f.menne@gmail.com; ingo@kth.se

© Springer International Publishing Switzerland 2016

F. Oppenheimer, J.L. Medina Pasaje (eds.), *Languages, Design Methods, and Tools for Electronic System Design*, Lecture Notes in Electrical Engineering 361,
DOI 10.1007/978-3-319-24457-0_7

have grown to such an extent that they can be considered affordable small-scale supercomputers. But the underlying architecture exhibits many intricacies, making it difficult to exploit. For instance, in order to reach maximum performance it is paramount that the GPGPUs's registers, on-chip memories, and caches are used efficiently, but optimizing the usage of one resource often has a negative impact on another. Moreover, the convoluted addressing schemes required for distributing data across the threads are mechanical, tedious, and error-prone to manage manually. Hence, to manually write applications that are both correct and efficient when executed on a GPGPU is an extremely challenging and error-prone task.

Although there exist several frameworks for elevating the task of GPGPU programming, they are all based on programming methodologies that hinder the use of automated tools for tasks such as verification, testing, and design space exploration. To mitigate these issues we present in this chapter a novel software synthesis tool—called f2cc[1]—that generates GPGPU code from applications which are represented as *abstract formal models*. These models have a solid formal foundation based on the theory of models of computation [15] and are devoid of low-level details regarding implementation and target architecture, which raises the level of abstraction for the system developer and enables the use of formal system design tools. In this case we use ForSyDe for modeling the applications. Hence f2cc promotes an application design flow that is "correct by construction" [8] by allowing the system developer to focus on *what* the system is meant to do rather than *how*, which lowers the development cost. Most importantly, f2cc enables system developers to take advantage of GPGPUs without needing to have extensive and in-depth knowledge about the underlying architecture.

The chapter makes the following contributions:

- We present a novel software synthesis tool (f2cc) that is capable of generating GPGPU code from abstract formal models based on the synchronous model of computation. Using a formal framework for application design enables the potential to perform verification, testing, and design space exploration in an automated fashion. Other advantages of the tool include:
  - *Modeling framework independence.* f2cc provides a flexible XML+C input format and frontend support which can be extended to support models created using different formal modeling methodologies.
  - *Adaptive and stand-alone code.* The GPGPU code produced by f2cc adapts itself to the properties of the graphics card at runtime, and does not depend on any proprietary libraries in order to be compiled or executed.
  - *Flexible data type support.* f2cc allows the developer to use custom-made `struct`s as data types in the models, thus facilitating the application design.

[1] Source code is available at http://forsyde.ict.kth.se/trac/wiki/ForSyDe/f2cc.

- We describe the methods and algorithms devised for f2cc, including an $O(n)$ algorithm for finding a process schedule for synchronous models containing feedback loops.
- We present experiments that demonstrate the correctness and efficiency of f2cc for GPGPU code synthesized for a Mandelbrot generator and an industrial-scale image processor. Compared against the performance of a hand-written CPU version, the GPGPU code generated by f2cc yielded a speedup of 28× when executed on a graphics card equipped with 96 cores.

The rest of the chapter is organized as follows. Section 7.2 briefly describes the GPGPU platform and introduces ForSyDe, the formal modeling methodology currently supported by f2cc. Section 7.3 explains the software synthesis process, the techniques and methods applied, and its current limitations. Section 7.4 gives the results from the experiments that were performed to validate the tool. Section 7.5 covers related work and discusses existing frameworks which elevate the task of GPGPU programming. Lastly, Sect. 7.6 concludes the chapter and Sect. 7.7 lists future work.

7.2 Background

7.2.1 GPGPUs

GPGPUs are enhanced versions of GPUs [9, 18], which are processing units specifically designed for rendering image frames. As image rendering is generally a parallel process where pixels can be generated independent from one another, the GPU quickly evolved into a massively data-parallel platform. Recognizing this vast computational resource, program developers urged the manufacturers to augment the GPU with functionality that would allow execution of applications written in general-purpose programming languages such as C. When adapted to GPGPUs applications have often yielded a significant performance increase, at times reaching orders of magnitudes in speedup [10].

A well-known family of GPGPUs is CUDA [9, 14, 17, 18], which is developed by NVIDIA. The CUDA platform, as illustrated in Fig. 7.1, consists of clusters of *streaming multiprocessors* (SMs) which are connected to a dedicated DRAM commonly referred to as the *global memory*. Each SM contains 8 *streaming processors* (SPs) or *CUDA cores* which share the same fetch/dispatch unit, register file, and instruction cache. The SM also consists of a set of various on-chip memories: a *shared memory*, sometimes denoted as *scratchpad memory*, which is an application-controlled cache; a *constant cache*, which retains constant values; and a *texture memory*, which is used to cache neighboring cells in a 2D data matrix. With the DRAM bandwidth usually being the main performance bottleneck, these caches are used to reduce the amount of traffic to and from the global memory.

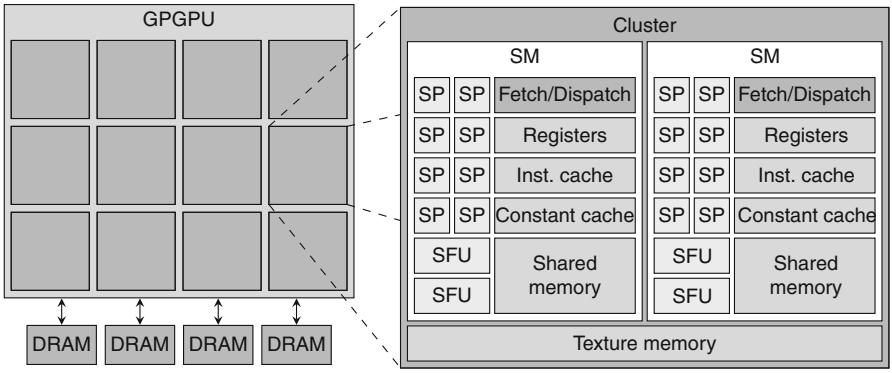


Fig. 7.1 Overview of the NVIDIA CUDA platform [17, 18]

After having copied the input data to the global memory, the GPGPU is accessed through a *kernel invocation* which spawns a set of threads to be executed on the GPGPU. These threads are bundled into *thread blocks*, which in turn are allocated onto the SMs. A small set of threads is then randomly selected from each thread block for execution on the SPs. Thread-context switches are made with virtually zero overhead, and provided that there is an abundance of threads the GPGPU can hide long latency operations through continuous thread switching. This makes the GPGPU a *throughput-oriented architecture* [9]. All thread blocks allocated to an SM share the same register file and other resources such as caches. This means that if a thread block uses too much of any resource, the maximum number of residential thread blocks per SM will be reduced. Fewer thread blocks means fewer threads to swap in and out to hide long latency operations, which in turn decreases the performance. Since the caches are very limited—the sizes are in the order of tens of kilobytes—meticulous care must be taken to not claim too much of any cache per thread block, and make optimal use of the allotted slice. Hence the main challenges of exploiting GPGPUs are as follows:

- *Adapting the application to fit the data-parallel execution platform.* Even algorithms that are inherently parallel may need to be redesigned in order to avoid performance-hampering issues such as *thread divergence* [14], which may occur when the code contains branch instructions.
- *Determining how to layout the input data and thread configuration.* The data needs to be packaged in such a way that it can be accessed from a thread using its thread and thread block IDs. Thus, there should be a correlation between the data layout and the thread configuration.
- *Determining which GPGPU resources to use, and how, in order to achieve optimal performance.* The GPGPU contains several resources such as caches and on-chip memories that can greatly boost performance. However, it is not always clear how each can be used for a particular application, often forcing new algorithms to be considered.

- *Determining whether utilizing the GPGPU is beneficial.* Even if all performance-inhibiting problems related to the GPGPU itself are dealt with, it is still possible that the code runs *slower* on the GPGPU than on the CPU. For example, the CPU may be relatively more powerful than the GPGPU, or there may not be enough computational complexity in the kernel to sufficiently amortize the GPGPU overhead of moving data between the main RAM and the GPGPU RAM.

7.2.2 ForSyDe

ForSyDe (Formal System Design) [1, 19] is a formal design methodology for embedded systems. It consists of a set of libraries, currently available in Haskell and SystemC, that enable modeling of systems at a high level of abstraction where the functionality of a system is detached from its implementation. The libraries support several *models of computation* (MoCs), but in the context of this chapter only the *synchronous MoC* is considered. The synchronous MoC is based on the *perfect synchrony hypothesis* [3], which assumes that data propagation and process execution take zero time (i.e., processes produce their output values immediately as their inputs arrive). This assumption leads to a simple and elegant mathematical model that fits nicely with a large class of data flow applications and with the underlying mechanisms of the GPGPU platform. The synchronous MoC is also base for the family of synchronous languages like Esterel [4] and Lustre [11], for which mathematical methods exist for performing verification and testing. Another similar modeling framework is StreamIt [21], where program hierarchy is modeled using predefined structures.

Systems are modeled in ForSyDe as hierarchical concurrent process networks, where *processes* communicate by means of *signals* (see Fig. 7.2). Processes are created using predefined *process constructors* that take *side effect-free functions* and *values* as arguments. This concept of process constructors leads to a clean separation between *communication* and *computation*: communication and model of computation is expressed by the process constructor; and computation is specified by the arguments of the process constructor. For example, Fig. 7.3a shows *mooreSY*, a process constructor to create a Moore finite state machine process belonging to the synchronous MoC. As arguments, *mooreSY* takes two functions `ns` and `o` and a value `s`: the function `ns` specifies the calculation of the next state; the function `o` specifies the calculation of the output value; and the value `s` specifies the initial state. ForSyDe process constructors can be divided into three categories

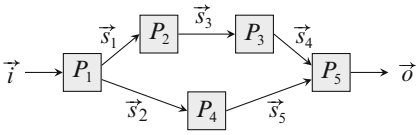


Fig. 7.2 Example of a ForSyDe model

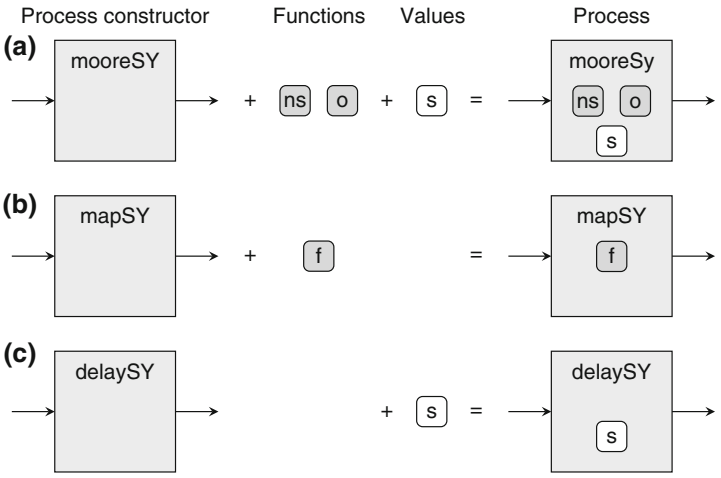


Fig. 7.3 A ForSyDe process constructor takes functions and values as arguments to form a process of a particular model of computation. Process constructors can be grouped into three different categories: sequential (*mooreSY*), combinational (*mapSY*), and delay process constructors (*delaySY*)

as illustrated in Fig. 7.3: sequential (*mooreSY*), combinational (*mapSY*), and delay process constructors (*delaySY*). These categories exist in all models of computation.

This separation of concerns is exploited when writing the ForSyDe models to text files. Using GraphML—the input format of f2cc (see Sect. 7.3.1)—the hierarchical structure of the process network is expressed in XML, and the computation is given as C code. Hence, the description of the structure is separated from the description of the computation. We want to point out that other formalisms that support the synchronous MoC, and provide a similar separation of communication and computation as ForSyDe, can be used in conjunction with f2cc as described in Sect. 7.3.1.

7.3 Synthesis Process

To synthesize ForSyDe process networks into a target implementation, an implementation technique obeying the ForSyDe semantics is required for (1) each process constructor, (2) the arguments of each process constructors, (3) the process network. However, this alone is in general not sufficient to yield an efficient implementation. Thus f2cc also identifies optimizations that can be applied to the model.

f2cc operates by first parsing an input file containing the model and converted into an internal model representation. Then a series of semantically preserving optimizations are applied, and lastly the model is synthesized into code. This process is also illustrated in Fig. 7.4. We will begin by discussing the input format, and then proceed with examining the internals of f2cc (more details are available in [13]).

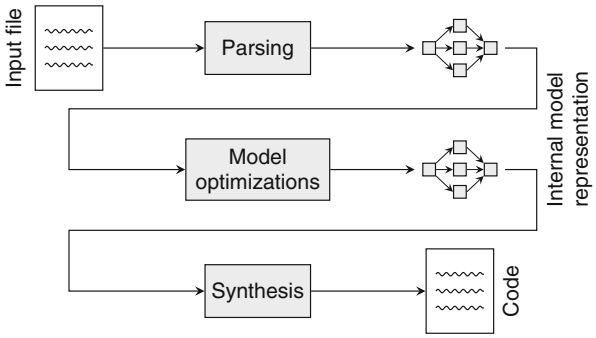


Fig. 7.4 Overview of the synthesis process

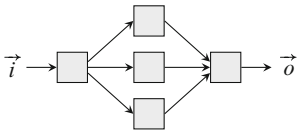


Fig. 7.5 Illustration of the model declared in Listing 7.1

7.3.1 Input Format

Once a model has been designed, it is passed to f2cc in the form of a *GraphML file*. Similar output can be generated from ForSyDe-SystemC using introspection [1], and converting it to GraphML is trivial. GraphML [5] is a standardized format-based XML in which graphs can be represented in a formal manner, and allows the process functions to be provided as data annotated to the nodes. The process functions are defined as side effect-free C functions, meaning they must not depend on any external state such as global variables or dynamically allocated memory. An example of such an GraphML file is available in Listing 7.1, whose model is illustrated in Fig. 7.5. Note that the input file contains no GPGPU-related information, thereby completely hiding any implementation-specific details about the target platform from the developer. Moreover, the input format does not require data types to be specified for signals and processes which do not have a C function as argument. Instead, the data types for these will be automatically inferred by f2cc during synthesis (see Sect. 7.3.4). This makes for a very versatile format that allows models to be created using any formal modeling framework, provided the models can be converted into the expected input format and hold the same semantic meaning. Since the format is human-readable the input files can even be written by hand. If desired, f2cc can also be extended with additional frontends to support for another input format.

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml>
 <graph id="test" edgedefault="directed">
  <node id="in">
   <data key="process_type">InPort</data>
   <port name="out" />
  </node>
  <node id="out">
   <data key="process_type">OutPort</data>
   <port name="in" />
  </node>

  <!-- Processes -->
  <node id="inc">
   <data key="process_type">ParallelMapSY</data>
   <data key="procfun_arg">
    <![CDATA[
    int func(const int arg) {
      return arg+1;
    }
    ]]>
   </data>
   <data key="num_processes">3</data>
   <port name="in" />
   <port name="out" />
  </node>

  <!-- Signals -->
  <edge source="in" sourceport="out"
        target="inc" targetport="in" />
  <edge source="inc" sourceport="out"
        target="out" targetport="in" />
 </graph>
</graphml>
```

Listing 7.1 Example of an input file to f2cc

7.3.2 Model Optimizations

In order to take advantage of the parallel nature of GPGPUs, the model needs to exhibit a certain level of data parallelism which can either be declared *implicitly* or *explicitly*. Implicit data parallelism is declared through a network of processes, known as a *data-parallel component*, while explicit data parallelism is declared via a single processes that semantically entail the functionality of entire data-parallel components.

There are many patterns of data parallelism. One such pattern is a data-parallel component that accepts an input array, applies one or more functions on every element or non-overlapping range of elements, and produces an array as output (see Fig. 7.6a). While simple, it is an important and powerful pattern that allows modeling of many embarrassingly parallel problems. We call this the *split-map-*

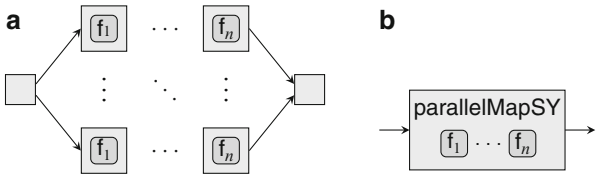


Fig. 7.6 The split-map-merge pattern. (**a**) Implicit declaration; (**b**) Explicit declaration

merge pattern: first, the array is *split* into multiple data sets, then one or more functions are *mapped* onto each data set, and lastly the results are *merged*. We have devised a special process constructor called *parallelMapSY* (see Fig. 7.6b) for explicit declaration of this pattern (which is equivalent to StreamIt's *splitjoin* construct), and support for exploiting it for efficient execution on GPGPUs is already available in f2cc. Our tool is also capable of combining chains of *map* processes into a single *map* process in order to reduce the amount of function calls, which we refer to as *process coalescing*.

Since discovering explicitly declared data parallelism is trivial (the data-parallel component is contained in a *single* process), the challenge lies in detecting implicitly declared data parallelism where a *cluster* of processes needs to be combined into a data-parallel component. For the split-map-merge pattern, this is done using an $O(n^2)$ depth-first algorithm which searches for pairs of *split* and *merge* processes. For a given pair, it then checks whether the data flow is contained between the two processes, and whether the intermediate processes between the *split* and *merge* processes consist of chains of *map* processes only. Once identified, the implicitly declared data-parallel components are replaced by single processes of the type which corresponds to the explicit declaration of the patterns (e.g., a data-parallel component arranged as the split-map-merge pattern will be replaced by a *parallelMapSY* process). This simplifies the later process schedule and code generation stages as each such process will constitute a complete and separate GPGPU kernel. It is possible to add support for exploitation of explicitly declared patterns of data parallelism while leaving out discovery of implicit declarations. In such instances, models containing implicit declarations will still be synthesized, however, the data-parallel component will be executed sequentially on the CPU instead of in parallel on the GPGPU.

7.3.3 Process Schedule Generation

As order of execution has an impact on the final output, a process schedule must adhere to the effects of the perfect synchrony hypothesis (i.e., that process execution and data propagation between processes take zero time). Finding such a schedule for sequential models is straight-forward—one just needs to traverse the model along its signals—but diverging data flows and feedback loops complicates this task.

```
function FindSchedule(M) returns schedule for model M
  schedule ← empty list; queue ← empty queue
  visited_G ← empty set
  for each output signal S of M do
    add process of S to head of queue
  while queue is not empty do
    visited_L ← empty set
    P ← head of queue; remove head from queue
    {p_schedule, ip} ← FindPartialSchedule(P, visited_G,
      visited_L, queue)
    if ip = "at beginning'' then
      insert p_schedule before head in schedule
    else
      insert p_schedule after process ip in schedule
    add visited_L to visited_G
  return schedule

function FindPartialSchedule(P, visited_G, visited_L, queue)
  if P ∈ visited_G then
    return {empty list, P}
  if P is a delay element then
    add preceding process of P to end of queue
    return {P, "at beginning''}
  schedule ← empty list
  ip ← "at beginning''
  if P ∉ visited_L then
    add P to visited_L
    for each preceding process O of P do
      {p_schedule, new_ip} ← FindPartialSchedule(O,
        visited_G, visited_L, queue)
      append p_schedule to schedule
      if new_ip ≠ "at beginning'' then
        ip ← new_ip
    append P to schedule
  return {schedule, ip}
```

Listing 7.2 Process scheduling algorithm

Listing 7.2 shows the algorithm which was devised for f2cc. It is based on a recursive depth-first search approach: starting from the model outputs, each process *P*is visited by traversing the model in the reverse data flow direction until no further traversing is possible (if the traversal was done in the forward data flow direction, then no schedule would be generated for models with no inputs). Partial schedules are then built and concatenated until the entire model has been traversed, and a set of visited processes is maintained in order to avoid redundant search and provide termination when feedback loops (i.e., cyclic data flow) is encountered. However, the synchronous MoC does not allow feedback loops without using a kind of delay element, and the placement of the this element within the loop affects the final

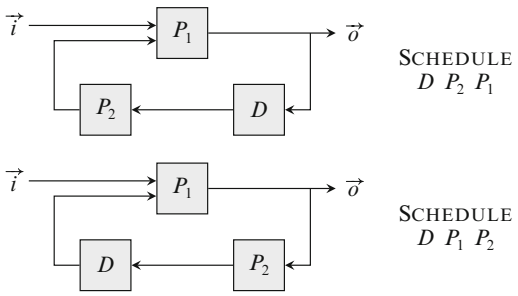


Fig. 7.7 Examples of two models with corresponding process schedules

schedule (as illustrated in Fig. 7.7). In this context, a delay element is a process that for an input sequence $\langle v_1, \ldots, v_n \rangle$ shifts the sequence in time by inserting an initial delay value s, thus producing $\langle s, v_1, \ldots, v_n \rangle$ (in ForSyDe this element is implemented using the *delaySY* process constructor). Our scheduling algorithm handles these situations by effectively acting as if the inbound edges to the *delaySY* processes had been removed. Using data structures that can be accessed in constant time, the algorithm finishes in $O(n)$ time.

7.3.4 Signal Data Type Inference

Signals are the vessels in the model through which data is propagated from one process to another. It is therefore appropriate to retain the notion of signals by implementing them as data containers in the synthesized code, typically as either global or local C variables. However, the data types of the signals are not immediately available from the formal model as they are only explicitly specified as part of the C functions, which only appear in the *map* processes. Hence the signals connected to other processes such as *delay*, *split*, and *merge*, the data types have to be automatically inferred. In f2cc this is done using an algorithm that recursively traverses the model until signal connected to a *map* process is found. This information is then propagated backwards to the original signal, and hence the data types ripple from signal-to-signal across the model as shown in Fig. 7.8. By caching the data type found for each signal, the algorithm takes $O(n)$ time to find the data types of all signals in a model. Failing to infer the data type for a signal indicates that the model is invalid, which is also reported by f2cc.

7.3.5 GPGPU Code Generation

In Listing 7.3 we provide the CUDA code generated by f2cc using the GraphML file given in Listing 7.1 as input. For each data-parallel component, which at this stage will have been converted into a single-process equivalents, f2cc will generate a set of

```
__device__ int finc_func1(const int arg) {
  return arg+1;
}

__global__ void finc_kernel(const int* in, int* out, int offset) {
  unsigned int gi = (blockIdx.x * blockDim.x + threadIdx.x) + offset;
  extern __shared__ int in_cached[];
  if (gi < 3) { // Prevents out-of-bound threads from executing
    int in_i = threadIdx.x * 1; int global_in_i = gi * 1;
    in_cached[in_i + 0] = in[global_in_i + 0];
    out[gi] = finc_func1(&in_cached[in_i]);
  }
}

void finc_kernel_wrapper(const int* in, int* out) {
  int* d_in; int* d_out; struct cudaDeviceProp prop;

  // Get GPGPU device information
  cudaGetDeviceProperties(&prop, 0);
  int max_t_per_b = prop.maxThreadsPerBlock;
  int smem_per_sm = (int) prop.sharedMemPerBlock;
  int full_utc = max_t_per_b * prop.multiProcessorCount;

  // Prepare device and transfer input data
  cudaMalloc((void**) &d_in, 3 * sizeof(int));
  cudaMalloc((void**) &d_out, 3 * sizeof(int));
  cudaMemcpy((void*) d_in, (void*) in, 3 * sizeof(int),
                    cudaMemcpyHostToDevice);

  // Execute kernel
  struct KernelConfig c;
  if (prop.kernelExecTimeoutEnabled) {
    int num_t_left = 3; int offset = 0;
    while (num_t_left > 0) {
      int num_t_exec = num_t_left < full_utc ? num_t_left : full_utc;
      c = calculateBestKernelConfig(num_t_exec, max_t_per_b, 1 * sizeof(int),
                                    smem_per_sm);
      finc_kernel<<<c.grid, c.threadBlock, c.sharedMemory>>>(d_in, d_out,
                                                             offset);
      int num_executed_threads = c.grid.x * c.threadBlock.x;
      num_t_left -= num_executed_threads;
      offset += num_executed_threads;
    }
  }
  else {
    c = calculateBestKernelConfig(3, max_t_per_b, 1 * sizeof(int),
                                  smem_per_sm);
    finc_kernel<<<c.grid, c.threadBlock, c.sharedMemory>>>(d_in, d_out, 0);
  }

  // Transfer result back to host and clean up
  cudaMemcpy((void*) out, (void*) d_out, 3 * sizeof(int),
                    cudaMemcpyDeviceToHost);
  cudaFree((void*) d_in);
  cudaFree((void*) d_out);
}

void executeModel(const int* in1, int* out1) {
  // Declare and alias signal array variables with model input/output arrays
  const int* vmodel_in_to_inc_in = in1;
  int* vinc_out_to_model_out = out1;

  // Execute processes
  finc_kernel_wrapper(vmodel_in_to_inc_in, vinc_out_to_model_out);
}
```

Listing 7.3 CUDA code generated for the input file given in Listing 7.1. Note that the code has been manually edited and shortened in order to fit this chapter

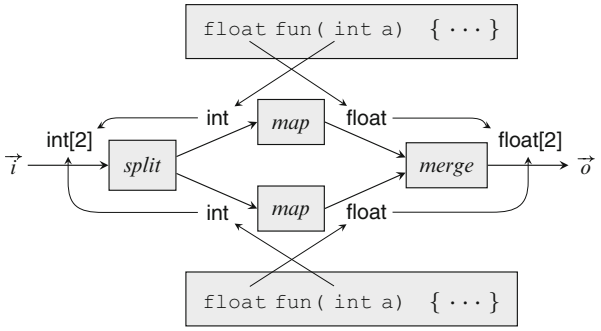


Fig. 7.8 Example of how the data types propagates along the signals

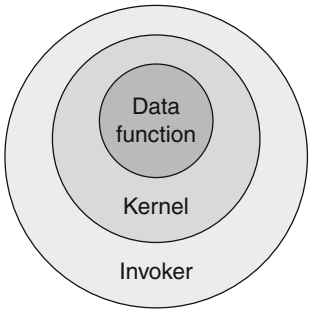


Fig. 7.9 The function stack used by f2cc for executing functions on the GPGPU

wrapper functions (see Fig. 7.9). The C function that implements the computational part of the data-parallel component—we will from now on call this the *data function*—is wrapped by a *kernel function*. The kernel function is responsible for providing the input data based on the thread block and thread IDs, managing the shared memory, and preventing out-of-bound threads from executing. In the case of the *split-map-merge* pattern, utilizing shared memory is done by first copying all the data required by the data function from global memory to the shared memory, and then passing the appropriate pointer to the data function. The kernel function is then wrapped inside an *invoker* function, which manages memory transfers between the CPU and GPGPU and sets up the thread configuration. The thread configuration is decided at runtime such that the size of the thread blocks is the maximum size supported by the graphics card (since the number of concurrent thread blocks per SM is limited to 8 at a time, it is necessary to use as large thread blocks as possible in order to achieve optimal performance). However, if the generated code makes use of shared memory then the threads may require more shared memory than available, which reduces the number of thread blocks per SM and thus inhibits performance. To prevent this an algorithm is employed which incrementally decreases the thread block size and calculates the amount of unused shared memory for that size. This continues until either the amount reaches zero, or until the number of thread blocks per SM becomes greater than 8 (upon which the configuration with the least waste

is selected). Some GPGPU execution environments may also enforce a maximum execution time for each kernel invocation, and f2cc embeds additional code for handling such situations when generating the invoker function.

7.3.6 Process Execution and Data Propagation

Executing the processes is straight-forward: the code simply needs to invoke the processes' C functions (if the process is of such type) with the appropriate parameters according to the generated process schedule. Data propagation is then done via a set of C variables—one for each signal—which are passed as parameters to the C functions. Part of the future work will be to identify and remove redundant signal variables, which will reduce the number of signal-to-signal copying operations and thus increase performance. Delay element values are stored in `static` C variables as these need to be retained between model invocations. For signals consisting of multiple values, the tool builds the necessary arrays and manages the addressing such that each process gets the correct input value.

7.3.7 Limitations

So far we have focused on supporting discovery and exploitation of the split-map-merge pattern. Hence f2cc does not yet provide full support for all process types that are available in ForSyDe, but the process type support as well as the recognition and exploitation of additional patterns of data parallelism can be extended by defining new process types, adding recognition of the new process types in the frontends, and extending the backend to synthesize the appropriate C or GPGPU code for each process type.

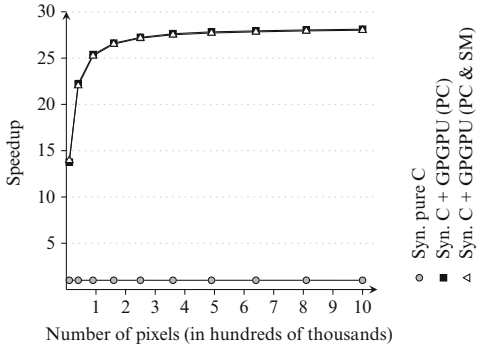
The synthesized GPGPU code also does not make full use of all available CUDA resources. Currently only the shared memory is considered, but this is simply because the potential resource usage is dependent on the pattern of data parallelism being exploited. In the case of the split-map-merge pattern, there is little or no gain in using the shared memory, or any other resource for that matter. Hence, these resources can be put to better use when additional patterns are available.

Another significant drawback is that no cost analysis is currently performed of whether it is actually beneficial to offload parallel computations onto the GPGPU. This means that, depending on the performance of the GPGPU and CPU, the generated CUDA code may run *slower* than if had been executed sequentially on the CPU.

a

Problem size (pixels)	Execution time (s)			
	Pure C impl.		C + GPGPU impl.	
	HW	Syn.	PC	PC & SM
10,000	1.33	1.33	0.10	0.10
40,000	5.30	5.30	0.24	0.24
90,000	11.92	11.91	0.47	0.47
160,000	21.19	21.19	0.80	0.80
250,000	33.09	33.10	1.21	1.22
360,000	47.66	47.66	1.72	1.73
490,000	64.87	64.86	2.33	2.34
640,000	84.72	84.72	3.03	3.04
810,000	107.23	107.21	3.82	3.84
1,000,000	132.39	132.42	4.71	4.73

Maximum measured standard deviation: 2.53%



b

Problem size (pixel domains)	Execution time (s)				
	Pure C impl.		C + GPGPU impl.		
	HW	Syn.	Basic	PC	PC & SM
1,000,000	0.38	0.40	0.10	0.08	0.09
2,000,000	0.77	0.81	0.15	0.11	0.13
3,000,000	1.15	1.21	0.21	0.14	0.17
4,000,000	1.53	1.62	0.26	0.17	0.21
5,000,000	1.92	2.02	0.31	0.21	0.25
6,000,000	2.30	2.42	0.36	0.24	0.29
7,000,000	2.68	2.82	0.41	0.27	0.33
8,000,000	3.06	3.22	0.47	0.30	0.37
9,000,000	3.45	3.62	0.52	0.34	0.41
10,000,000	3.83	4.03	0.57	0.37	0.45

Maximum measured standard deviation: 0.86%

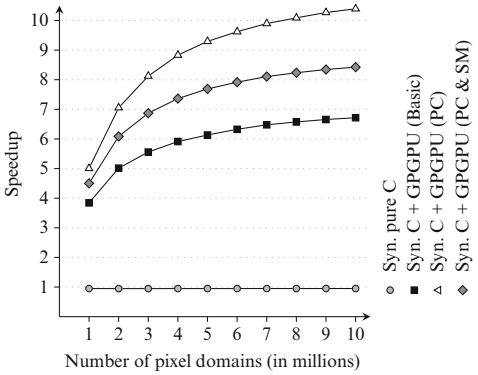


Fig. 7.10 Experimental data. HW stands for *hand-written*, and "Syn." refers to the code generated by f2cc, where PC and SM indicates whether process coalescing or shared memory on the GPGPU was used, respectively. (**a**) Test results from the Mandelbrot model; (**b**) Test results from the image processing model

7.4 Experiments

To validate the correctness and efficiency of f2cc, the tool was applied on models derived from two applications: a Mandelbrot generator, and an industrial-scale image processor. For each model, a pure C implementation of the final code and multiple implementations where the data-parallel components are executed on the GPGPU were generated and evaluated (see Fig. 7.10). The output and performance of the synthesized C code was compared with a hand-written C version which was executed by a single thread on the CPU. The C code and GPGPU code was compiled using *g++* v.4.6.1 and *nvcc* release 3.2 v0.2.1221, respectively, with all optimizations disabled. The test cases were executed on an Intel Core i7-2600 at 3.40 GHz, 16 GB DDR3 RAM at 1333 MHz, and an NVIDIA Quadro 600 with 96 CUDA cores, 1 GB DDR3 RAM. Each test case was run 10 times and then an arithmetic mean average was calculated from the results.

7.4.1 Mandelbrot Tests

Generating Mandelbrot images is a task exhibiting an abundance of data parallelism. Each pixel coordinate is converted into a corresponding coordinate within a rectangular coordinate window in the complex plane. From the complex coordinate an integer value is computed which determines to whether the coordinate is part of the Mandelbrot set. In these tests, the window was bounded by $(-\frac{1}{4}, -\frac{1}{4})$ and $(\frac{1}{4}, \frac{1}{4})$. Its model consists of a single data-parallel component, and when expressed using *parallelMapSY* the model shrinks to a single process. The performance results of the synthesized C and GPGPU code are given in Fig. 7.10a. We see that the synthesized C code performs equally with the hand-written C version, and the synthesized C + GPGPU code performs 28× better. The relatively low speedup for small input data sizes is due to the restricted amount of computations which can be offloaded to the GPGPU. As the input data size increases, so does the extent to which the GPGPU overhead can be amortized. Since there is very little input data reuse and no data sharing, using shared memory has no impact on the performance. The output of the synthesized C code was exactly equal to that of the hand-written version, but for the GPGPU code the integer values were slightly different for some coordinates. We believe this discrepancy to be caused by the floating point units whose architecture differ between the GPGPU and CPU.

7.4.2 Image Processing Tests

The second model was derived from an existing industrial-scale image processor application provided by XaarJet AB, a company specializing in piezoelectric drop-on-demand ink-jet printing. At its core, the model consists of a single data-parallel component composed of 3 data-parallel segments. Using the *parallelMapSY* process constructor and process coalescing, this model also shrinks to a single process. The details of the C functions will not be covered as not to disclose any industry secrets. The performance results are given in Fig. 7.10b. Again, the synthesized C code is on par with the hand-written version, and the synthesized C + GPGPU code is 10× faster. This relatively low speedup is due to lack of computational complexity in the model, and the continued slope indicates that greater speedup is achievable with even larger problem sizes. Furthermore, as the input data size per thread is much greater than in the Mandelbrot model, the performance of the synthesized GPGPU code is reduced when the shared memory is used since doing so will limit the number of thread blocks that can simultaneously reside in an SM, which in turn lowers performance. Like with the Mandelbrot tests, the synthesized code produces slightly different output when executed on the GPGPU compared to the CPU. Since floating point operations are involved, we again believe the differing architectures of FPUs between the CPU and GPGPU to be the cause.

7.5 Related Work

Existing GPGPU programming frameworks can generally be divided into three categories: *declarative-based frameworks*, where code to execute on the GPGPU is marked by annotations; *library-based frameworks*, where the core is implemented as programming libraries; or *domain-specific languages* (DSLs), where the framework is embedded into an existing programming language.

Declarative-based frameworks include hiCUDA [12] and OpenMP-to-GPGPU [16]. In hiCUDA parallelizable C code is annotated with pragma directives which control dynamic memory allocation, thread configuration, work distribution per thread over loops, and more. The hiCUDA compiler then processes the code to generate GPGPU kernels based on the annotations. The framework therefore relieves the developer from having to produce the data addressing schemes, handle the CPU-GPGPU data transfers, and manage the shared memory. Consequently, hiCUDA relies on the developer to identify and tweak the code for execution on the GPGPU. In OpenMP-to-GPGPU the existing OpenMP pragma notations are used to identify parallelizable code, but these miss the information about thread blocks and shared memory. In both cases, the frameworks completely lack a formal foundation and are thus unsuitable for automated verification and testing.

Library-based frameworks include Thrust [2] and SkePU [7], which are both implemented in C^{++} and provide a set of *skeletons* (a skeleton is akin to the notion of process constructors used in ForSyDe, see Sect. 7.2.2). The developer provides the computation part to the skeletons, and the skeletons then decide the appropriate thread configuration, memory management, and other execution-related details. Unlike Thrust, SkePU is also capable of generating code for multi-core CPUs, OpenCL, and single-threaded C code. But although the use of skeletons provides a more formal base than pragmas, they are not based on a well-defined model of computation, and can therefore not be analyzed using existing mathematical tools. Moreover, the skeletons do not extend into the rest of the application.

Two GPGPU-oriented DSLs, both embedded in Haskell (a purely functional programming language), include Accelerate [6] and Obsidian [20]. Accelerate also uses the notion of skeletons by providing a collection of arrays and array operations that can be offloaded on a GPGPU. In order to compile into an application that can be executed on a GPGPU, Accelerate comes with a Haskell-to-CUDA compiler which translates Accelerate-based Haskell programs into CUDA-annotated C code. Obsidian is similar to Accelerate but instead provides a collection of *combinators* that allow array functions to be converted into GPGPU kernels. Through the combinators, the developer gains access to use of the shared memory and insertion of synchronization barriers, but this requires the developer to know when and how to use the combinators in order to match the underlying architecture of the GPGPU. Moreover, neither is based on a well-defined MoC, which again inhibits automated verification and testing.

7.6 Conclusion

In this chapter we have presented f2cc, a software synthesis tool which is capable of synthesizing abstract formal models based on the synchronous model of computation into GPGPU code. Unlike existing frameworks which elevate the task of GPGPU programming, f2cc operates on abstract formal models which enables the potential to apply automated tools on the applications for verification, testing, and design space exploration. Through experimental validation, we have shown that the tool produces correct and high-performing GPGPU code from its input models.

7.7 Future Work

Future work will primarily focus on integrating the results of Attarzadeh Niaki [1] to achieve a completely automated flow from ForSyDe-SystemC to GPGPU code. Another consideration is more efficient signal handling methods to eliminate redundant memory transfers between execution of separate data-parallel components.

In addition, the number of recognizable and exploitable patterns of data parallelism that can be executed on the GPGPU will be expanded. For example, a common pattern is *reduction data parallelism*, which is illustrated in Fig. 7.11. Generating efficient implementations of reduction patterns is more difficult compared to split-map-merge patterns because it requires more efficient use of the shared memory. Moreover, a naïve implementation will also lead to so-called *thread divergence* which hampers performance when executed on the GPGPU. Another pattern of data parallelism is a variant of the split-map-merge pattern where parts of the input data is used by multiple processes. A common instance is where the input data is formed as a 2-dimensional array which is then divided into slices that partially overlap one another (see Fig. 7.12). Efficient implementations of such patterns often require use of additional resources such as constant cache and texture memory.

Lastly, we also want to extend f2cc to make better judgement of when it is beneficial to use the GPGPU. Initial work has been done by Ungureanu [22] to

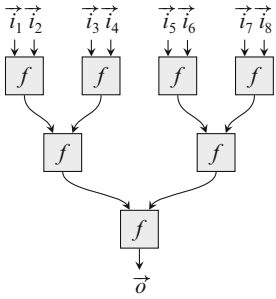


Fig. 7.11 Reduction data parallelism

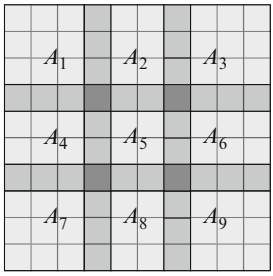


Fig. 7.12 A 2-dimensional input data set, where each slice A_1 through A_9 consists of 4×4 elements and partially overlaps with its neighboring slices

include the relative costs of executing a particular process on a specific target platform, but it is still at an experimental stage where the costs are computed and annotated by hand.

References

1. Attarzadeh Niaki, S.H., Jakobsen, M.K., Sulonen, T., Sander, I.: Formal heterogeneous system modeling with SystemC. In: Forum on Specification and Design Languages, FDL 2012, pp. 160–167, Vienna, Austria, September 2012
2. Bell, N., Hoberock, J.: Thrust: A productivity-oriented library for cuda. In: Wen-mei, W.H. (ed.) GPU Computing Gems, Jade edition, Chapter 26, pp. 356–371. Morgan Kaufmann, Los Altos, CA (2011)
3. Benveniste, A., Berry, G.: The synchronous approach to reactive and real-time systems. Proc. IEEE **79**(9), 1270–1280 (1991)
4. Berry, G., Cosserat, L.: The ESTEREL synchronous programming language and its mathematical semantics. In: Brookes, S., Roscoe, A., Winskel, G. (eds.) Seminar on Concurrency. Lecture Notes in Computer Science, vol. 197, pp. 389–448. Springer, Berlin (1985)
5. Brandes, U., Eiglsperger, M., Lerner, J.: GraphML Primer (June 2004). http://graphml.graphdrawing.org/primer/graphml-primer.html (last visited 2014-05-19).
6. Chackravarty, M.M.T., Keller, G., Lee, S., McDonell, T.L., Grover, V.: Accelerating haskell array codes with multicore GPUs. In: Proceedings of the 6th Workshop on Declarative Aspects of Multicore Programming (DAMP'11), pp. 3–14 (2011)
7. Dastgeer, U., Kessler, C.W., Thibault, S.: Flexible runtime support for efficient skeleton programming on hybrid systems. In: Proceedings of the International Conference on Parallel Programming (ParCo'11), Heraklion, Greece (2011)
8. Edwards, S., Lavagno, L., Lee, E.A., Sangiovanni-Vincentelli, A.: Design of embedded systems: formal models, validation, and synthesis. Proc. IEEE **85**, 366–387 (1997)
9. Garland, M., Kirk, D.B.: Understanding throughput-oriented architectures. Commun. ACM **53**, 58–66 (2010)
10. Garland, M., Le Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., Phillips, E., Zhang, Y., Volkov, V.: Parallel computation experiences with cuda. IEEE Micro **28**, 13–27 (2008)
11. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous dataflow programming language LUSTRE. Proc. IEEE **79**(9), 1305–1320 (1991)
12. Han, T.D., Abdelrahman, T.S.: hiCUDA: high-level GPGPU programming. IEEE Trans. Parallel Distrib. Syst. **22**, 78–90 (2011)

13. Hjort Blindell, G.: Synthesizing software from a ForSyDe model targeting GPGPUs. Master's thesis, KTH Royal Institute of Technology, School of Information and Communication, Stockholm, Sweden (2012)
14. Kirk, D.B., Wen-mei, W.H.: Programming Massively Parallel Processors. Morgan Kaufmann, Los Altos, CA (2010)
15. Lee, E.A., Sangiovanni-Vincentelli, A.: A framework for comparing models of computation. IEEE Trans. Comput.-Aided Design Integr. Circuits Syst. **17**(12), 1217–1229 (1998)
16. Lee, S., Min, S.-J., Eigenmann, R.: OpenMP-to-CUDA: a compiler framework for automatic translation and optimization. In: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'09), vol. 44, pp. 101–110 (2009)
17. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J.: Nvidia Tesla: a unified graphics and computing architecture. IEEE Micro. **30**, 39–55 (2010)
18. Nickolls, J., Dally, W.J.: The GPU computing era. IEEE Micro **30**, 56–69 (2010)
19. Sander, I., Jantsch, A.: System modeling and transformational design refinement in ForSyDe. IEEE Trans. Comput.-Aided Design Integr. Circuits Syst. **23**, 17–32 (2004)
20. Svensson, J., Claessen, K., Sheeran, M.: GPGPU kernel implementation and refinement using obsidian. In: Proceedings of the International Conference on Computational Science (ICCS'10), vol. 1, pp. 2065–2074 (2010)
21. Thies, W., Karczmarek, M., Amarasinghe, S.P.: StreamIt: a language for streaming applications. In Proceedings of the 11th International Conference on Compiler Construction, CC '02, pp. 179–196 (2002)
22. Ungureanu, G.: Automatic software synthesis from high-level ForSyDe models targeting massively parallel processors. Master's thesis, KTH Royal Institute of Technology, School of Information and Communication, Stockholm, Sweden (2013)

Chapter 8
A Framework for Distributed, Loosely-Synchronized Simulation of Complex SystemC/TLM Models

Christian Sauer, Hans-Martin Bluethgen, and Hans-Peter Loeb

Abstract Today's virtual prototypes model complex many-core platforms. In application domains such as network processing, they may comprise hundreds of processors, which makes simulation speed the key issue due to the single-threaded execution semantics of SystemC. We propose CoMix, the Concurrent Model Interface, for the distributed simulation of large-scale SystemC models. CoMix provides robust communication between simulator peers, enables their loose synchronization, and manages the overall life cycle. It is an overlay technology neither requiring modified simulators nor depending on a hosts' communication infrastructure. The CoMix framework is small (2k Lines of C++ Code) and easily deployable. We quantify its overhead on synthetic benchmarks and observe reasonable speedups for synthetic benchmarks as well as a large real-world example, e.g., 3.3X and 4X for a 4-peer simulation.

8.1 Introduction

Enabled by maturing standards, the availability of platform libraries, and wider tool support, SystemC (SC)-based simulation models are increasingly deployed early in the design cycle of System-on-Chip (SoC) platforms. Such models facilitate the development of embedded software for full, highly complex systems, as they abstract irrelevant details for faster simulation while providing sufficient insights into the interplay between software and hardware. This way, high-quality software can be developed sooner and more concurrently to a platform's hardware. In addition, these models may serve as entry points into exploration, design, and verification flows, because they capture the system intent in a functionally correct way [2].

C. Sauer (✉) • H.-M. Bluethgen • H.-P. Loeb
Cadence Design Systems, Munich, Germany
e-mail: sauerc@cadence.com

© Springer International Publishing Switzerland 2016

F. Oppenheimer, J.L. Medina Pasaje (eds.), *Languages, Design Methods, and Tools for Electronic System Design*, Lecture Notes in Electrical Engineering 361,
DOI 10.1007/978-3-319-24457-0_8

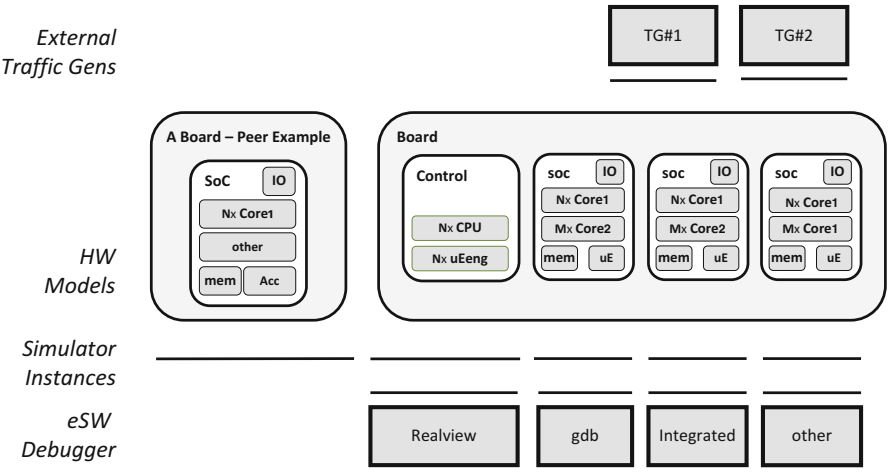


Fig. 8.1 Generalized SW development use case for a distributed simulation with heterogeneous debug and simulation tools

Contemporary SoCs are complex many-core platforms [3]. Especially network infrastructure, such as radio base stations or routers, may easily comprise multiple SoCs each with 10s–100s of processor cores along with memories, interconnect hierarchies, and various accelerator and IO modules. Models in this domain can instantiate 10s of thousands of SC objects. Their joint simulation with instruction-precise processor models makes the speed of the simulation a key issue. With fixed requirements on abstraction level (e.g., programmer's view) and modeling techniques (TLM—transaction level modeling), other ways are needed to improve simulation speed and to tackle the complexity of the models. Distributing the SystemC/TLM simulation into multiple parts that run in parallel, potentially on different simulation hosts, is a promising approach.

Yet, for the model to be widely usable, a suitable solution should support the generalized use case as in Fig. 8.1. A hierarchical simulation model is set up to run a multi-SoC simulation in a distributed fashion. Its parts run on different SC simulators and comprise heterogeneous cores which are to be debugged simultaneously with different tools, be it a core's native tool chain, standalone 3rd-party debuggers, or integrated multi-core debuggers. None of these tools nor the used communication infrastructure must monopolize the execution and block other tools and simulators from functioning. These requirements exclude prior solutions relying on IO virtualization of dedicated SoC interfaces [1] or on changes to the SC simulation engine [12, 15, 17].

We propose *CoMix*, the Concurrent Model Interface, as orchestrating infrastructure for the distributed simulation of large-scale SystemC models. CoMix provides robust, asynchronous communication between peers, enables their loose synchronization, and comprehensively manages the overall life cycle. It is a modular, vendor-independent overlay technology supporting the full range of SC and TLM communication primitives.

Before going into the key principles of our solution in Sect. 8.3, we overview SystemC and TLM briefly in Sect. 8.2. Section 8.4 details the implementation of the CoMix framework and discusses its interaction with the SystemC/TLM simulation libraries. Section 8.5 evaluates and characterizes CoMix using a set of synthetic benchmarks as well as a real-world virtual prototype. We compare our approach to related work (Sect. 8.6) and conclude on its features in Sect. 8.7.

8.2 SystemC and TLM

SystemC [9] is a system modeling language and C++ class library which adds distinct notions of hierarchy, concurrency, and simulation time to C++. A system is composed hierarchically from modules that communicate explicitly via ports/exports and channels. Its actual function is described within the modules as a collection of concurrent SC processes which explicitly synchronize on events (notify/wait). These processes are scheduled cooperatively. Execution is thread-safe as the SC scheduler runs them sequentially within a single OS thread.

TLM [9] is a modeling library on top of SC, which provides abstractions for the communication protocol and interfaces between SC modules. Modules may use sockets (sets of ports and exports) to exchange transactions between initiators and targets. Such exchange may be split into several phases or timing points, increasing the temporal resolution of the transfer. Blocking transfers have two timing points, while non-blocking transfers have at least four timing points. The former are modeled as a single function call that blocks the initiator's execution until the result is available, while the latter enable continued execution, potentially initiating further transactions before the first one completes (via callbacks, sequencing through the phase diagram, cf. Fig. 8.4).

TLM also introduces the concept of temporal decoupling, allowing processes to run ahead of the simulation time up to an upper limit, the quantum. At points of communication, a SC process may choose to first synchronize its local time with the global simulation time, i.e., to yield to other processes, or may continue its execution unsynchronized, maintaining a delta time. Synchronization guarantees correctness of an access, e.g., to a shared state. Unsynchronized continuation just accesses the current state accepting the temporal error associated with accessing that state too early or too late. Temporal decoupling is commonly used in the context of virtual platform simulations where the software stack does not depend on the low-level timing details of the hardware, which means the temporal error does not manifest functionally. Trading off simulation speed and accuracy, the error can be controlled by the value of the quantum, which depends on the application: A too large value may harm the system's function (e.g., trigger a software timeout), while a too small value yields frequently and slows down the simulation.

8.3 CoMix Fundamentals

CoMix is a modular collaboration infrastructure which allows heterogeneous SC simulators to concurrently execute a distributed SC/TLM model.

Partitioned SC/TLM Model As a prerequisite, a simulation model, i.e., a hierarchy of communicating SC modules, must be cut and grouped into parts for the individual simulators, as shown in Fig. 8.2. In the process, all connection cuts are assigned a unique identifier (cut id). Such a partitioning does not necessarily have to be along SC hierarchies [12, 13]. Yet, following natural boundaries of SoCs or IP subsystems will avoid hierarchy inconsistencies and maintain accessibility for tools. This may require a structural transformation of the original model. The result is a collection of SC modules for each part with open ports or sockets representing a cut. In a sequential simulation, these parts can be instantiated, connected, and simulated together, e.g., for verification purposes (Fig. 8.2, top).

In the distributed case, the individual parts are loaded into different simulators (Fig. 8.2, bottom). On each of the simulators a CoMix *peer* module is inferred and all open ports are bound to CoMix *connectors* either directly (TLM sockets) or via a channel (SC ports). Alternatively this may happen explicitly, coded as part of the SC netlist.

Network of Peers Before the distributed simulation starts, those peers that share at least one cut SC/TLM connection establish a direct TCP/IP link for the exchange of messages. Peers without communication requirements are not connected and do not synchronize.

Synchronization CoMix follows a loosely timed synchronization scheme that is similar to the concept of temporal decoupling in TLM. Each simulator advances its local time up to a configurable quantum, called the sync credit. Once the quantum has been reached (i.e., available credit has been consumed), synchronization with connected peers takes place, which means credit is granted to connected peers. The simulation halts if and only if there is insufficient credit available. Such a scheme preserves and exploits TLM's temporal decoupling semantics as each simulator may advance SC time locally at its own speed during sync intervals. As with TLM, the temporal error between peers can be controlled by the sync interval. Credits may be received any time, not just at the end of sync intervals. Thus, depending on the duration of the sync interval and the load distribution between peers, the slowest simulator can be expected to never stop its advancement of SC time, which effectively minimizes the synchronization overhead for the overall execution time.

Communication During the decoupled execution, communication between SC modules on different peers may take place. Such communication carries a time stamp that can be used to synchronize with a target's local simulation time. Recognizing the need for different, application-dependent schemes, CoMix encapsulates the handling of a connection's synchronization requirements within the associated pair of connectors. They may synchronize to the local time, so that a message is

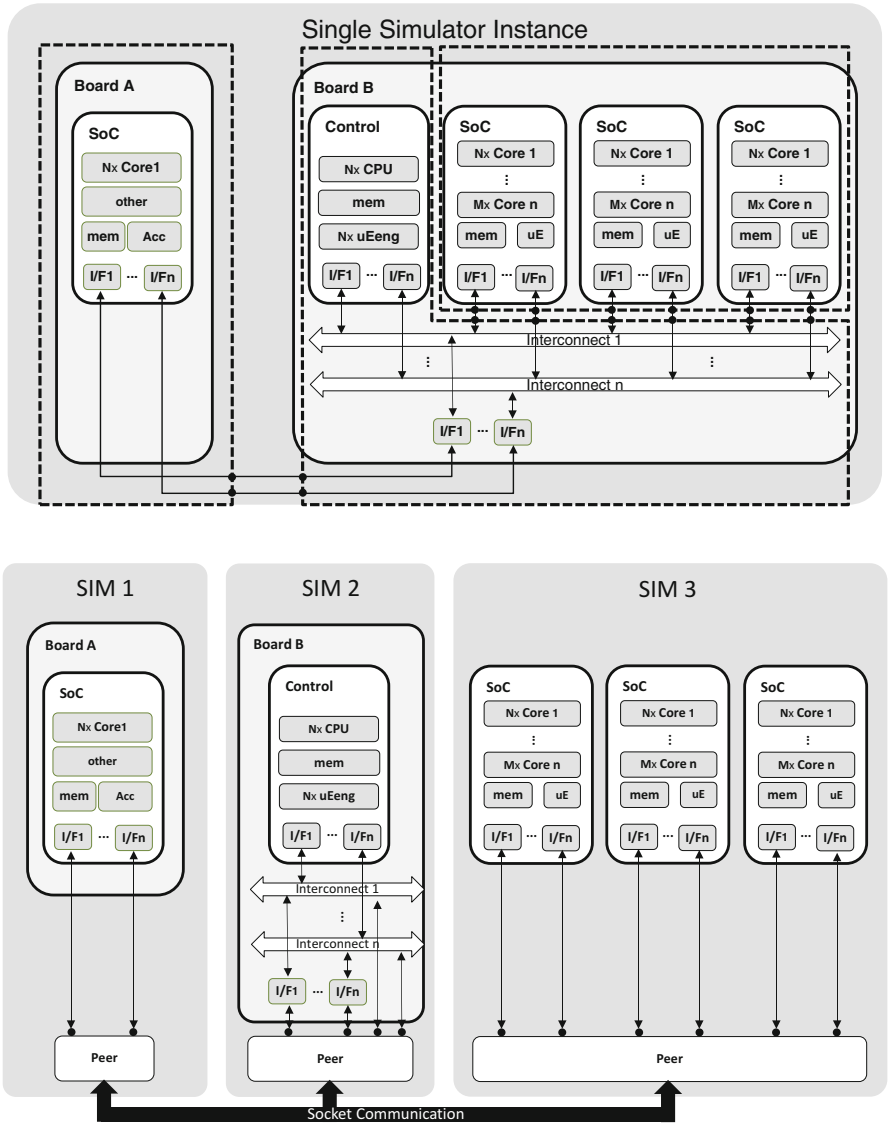


Fig. 8.2 A model (cf. Fig. 8.1) (*top*) is partitioned manually cutting SC connections, and distributed across three peers using CoMix (*bottom*), which handles the communication between cuts

not processed before its creation time, or may handle it immediately at the time of its reception. In both cases, the order of transactions within the same stream is maintained while independent streams may interleave differently towards the same target.

SC Support and Simulator Interaction CoMix supports the full range of SystemC communication primitives, i.e., communication via ports and channels as well as communication via TLM sockets. A current limitation is the lack of DMI support between sockets on different hosts. Life cycle management is controlled by the four simulator callbacks into the CoMix Peer. Interaction towards the simulator is required in only two cases: (1) if sync credit is lacking, the simulator is starved, and (2) in the event of SC communication, which is received asynchronously by CoMix and results in an *async_request_update()* call [9].

8.4 CoMix Framework

CoMix connects the distributed parts of a simulation, provides robust communication between peers, enables their loose synchronization, and comprehensively manages the overall life cycle. The framework is implemented in C++ and only relies on SC/TLM and Boost's ASIO library.

Figure 8.3 shows the main building blocks of the framework and their interaction. A single instance of the CoMix peer manages its enclosing simulation in the distributed setup. It comprises a multi-socket object, an asynchronous receive queue, and functions for message routing, life cycle management, and the synchronization with other peers. This peer is associated with a collection of CoMix connectors that are bound to the previously open SC ports/sockets of the model. These connectors translate SC communication into message sequences and vice versa. The peer itself

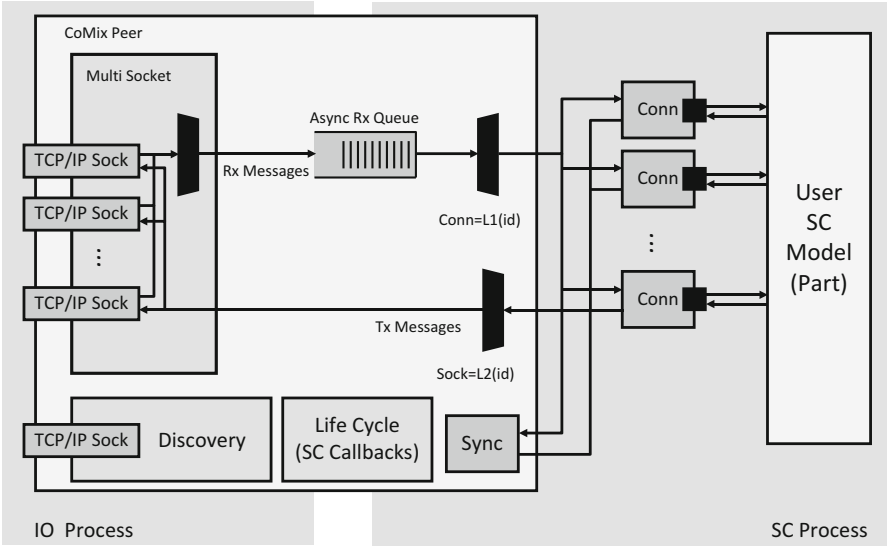


Fig. 8.3 Components of the CoMix framework

handles messages related to synchronization and life cycle management directly. Within the multisocket class an extra, shared OS thread is introduced which handles most IO operations asynchronously to SystemC executed in the main thread. Only outbound messages are sent synchronously.

8.4.1 CoMix Peer

Most of the CoMix function is encapsulated within the CoMix peer. As an SC object this class can be integrated into a model like any other module. All peers within one distributed simulation are identical.

During startup, peers discover each other and form a mesh as required for the connectivity of the simulation model. For this, the peers run a discovery protocol in which the peer started first becomes a super peer running on a specified listening address (IP, port). Others can connect to it, authenticate, and announce their local cut ids together with their own listening address. The super peer broadcasts this info to all its other connections. Upon reception of such a message a peer opens a direct and authenticated connection to the originating address, but only if they share a cut id. Once all local cut ids are associated with their remote counterpart, a peer's setting is considered sane and the connection to the super peer may be closed. Tables with remote and local cut ids are kept for routing messages locally to the socket connection (send) or the local CoMix connector (receive), respectively.

Since the reception of a message is asynchronous in a separate OS thread, it must be explicitly synchronized with the SystemC simulation. This is handled by the receive queue which guards accesses with locks and asynchronously notifies the kernel. In the event of written data a SC process is activated, which performs the lookup and forwards the message to the appropriate connector.

The peer also handles the synchronization of SC time between simulators. Listing 8.1 shows the pseudo code for one of the synchronization modes, the fully starved mode. After using up its own credit (6), a peer sends credit to all of its connected peers (7) and then starves the SC simulation completely within the inner loop (9) until it received sufficient credits from its peers. While SC is blocked, asynchronous reception and processing of messages must continue, hence the asynchronous receive queue is read periodically (12).

Listing 8.1 Fully-starved synchronization scheme

```
1
2  void sync_th() {
3
4    while ( !canSync( SIM_STOP) ) {
5
6      wait( sc_time( credit_ns , SC_NS) );
7      send_credit(ALL_PEERS, SYNC_CREDIT, credit_ns);
8
9      while (!canSync(QUOTA_SYNC)) {
10       if (canSync(SIM_STOP) ) break;
11       usleep(interval_us);
12       nb_recvMessage();
```

```
13      }
14
15      clearSync(QUOTA_SYNC);
16    }
17
18    send_credit(ALL_PEERS, STOP_CREDIT);
19    sc_stop();
20  }
```

Another synchronization mode is the delta-only mode, partly shown in Listing 8.2. In this case, the SC simulator is not starved but continues advancing time in delta cycles (12) while waiting for sync credit. This way transactions arriving late may still be processed within the past quota, which can reduce the temporal error at the initiator side. Explicit reads from the asynchronous receive queue are not required.

Listing 8.2 Delta-only synchronization scheme

```
3
4       ...
5
6       wait( sc_time( credit_ns, SC_NS) );
7       send_credit(ALL_PEERS, SYNC_CREDIT, credit_ns);
8
9       while (!canSync(QUOTA_SYNC)) {
10        if (canSync(SIM_STOP) ) break;
11        usleep(interval_us);
12        wait(SC_ZERO_TIME);
13      }
14
15      ...
```

The outer loop (4) handles the synchronization at the end of the simulation. Before a peer stops (19), it sends out stop credit to its peers (18), enabling them to stop as well. Further life cycle management is achieved by means of SystemC's simulation callbacks, which are forwarded to notify peers.

8.4.2 Connectors

CoMix connectors link open TLM and SC ports of a partitioned design with the CoMix messaging infrastructure. They are bound to their respective SystemC port or TLM socket and associated with the CoMix peer. An extensible set of CoMix connectors exists that can be categorized by:

1. Synchronization of inbound messages. Some connectors contain a payload event queue for inbound messages, which naturally handles their synchronization to the local SC time on a per connection and message type basis. For instance, *scsignal* value updates or *btransport* calls may be synchronized; *transportdbg* calls do not consume time and are not synchronized.
2. Handling of delta time. Outbound TLM connectors may synchronize a delta-time before sending a message or just annotate it. Similarly, inbound transactors may annotate future time as delta time on transactions instead of synchronizing it locally.

3. SC interfaces. Connectors for the different SC port types and TLM socket types are specialized from common bases. In some cases, standard-compliant protocol transformations are required, e.g., for transitioning through approximately timed communication, see Fig. 8.4.
4. Optimized return paths. Connectors are fully SC/TLM protocol compliant, which requires signaling back the result of a (potentially erroneous and delayed)

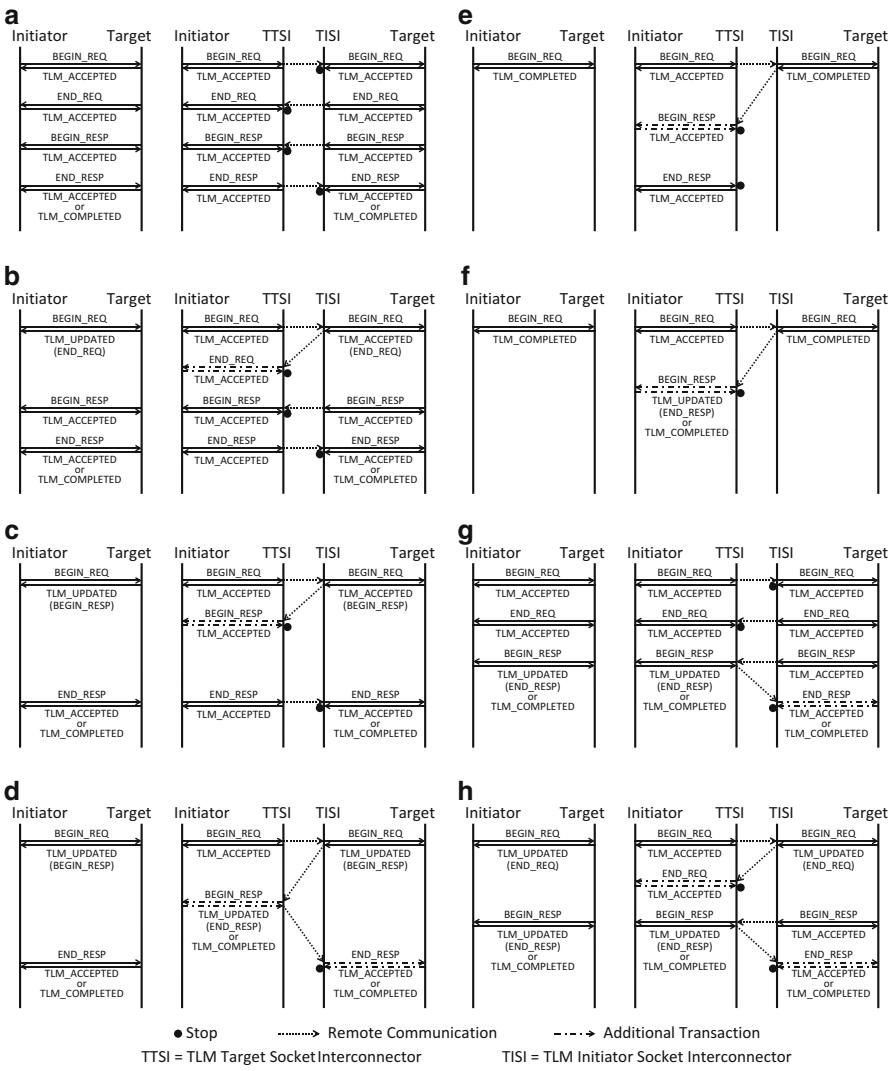


Fig. 8.4 Protocol conversion by nb-transport connectors. Variants a–h show different state transitions between initiator and target for the uncut case (*left*) and for the distributed case with connectors (*right*)

transaction to the initiator. While this cannot be avoided for, e.g., blocking read accesses, it may not be required in all applications. In case of writes, for instance, optimized connectors may skip the status response and instead assert on potential errors. This way, a blocking write call will never block its initiator.

Connectors were designed such that they can be created and configured dynamically by a factory and configuration infrastructure [14]. This enables setting their cut ids through a parameter interface from a suitable design description, which may also comprise the partitioned design.

8.4.3 CoMix Multisocket

The communication between peers is based on TCP/IP sockets which are accessed via the boost asio library. The CoMix multisocket holds a set of connections and a TCP acceptor. It also manages the shared OS thread for the asynchronous IO operations. Messages exchanged over socket connections are translated into boost property trees. Using this standard format ensures that arbitrary message types with widely differing content can be handled robustly in a generic way. At the lowest level of the socket IO library, functions are available for easy serialization of these data structures to character streams and vice versa.

8.4.4 Framework Characteristics

One design goal of our framework was to keep code complexity as low as possible by using standard libraries for both stability and maintainability. Although CoMix provides a powerful feature set, its complexity in terms of lines of code with 2k LoCs remains fairly low, cf. Table 8.1. The SC library, for instance, has 40X as much code. The framework is extensible and even supports, e.g., interfaces to non-SystemC tools, such as (remote) debuggers or traffic generators, by means of specialized peers and connectors.

Table 8.1 Code size of the CoMix framework compared to the SC/TLM library (as reported by cloc)

CoMix	Lines of code
Base	572
Peer	275
Connectors, btransport(), and signal	492
Connectors, nbtransport(), other	648
CoMix total	1.987
SystemC & TLM (2.3.0)	78.359

8.5 Case Study and Results

We apply the CoMix framework to the domain of packet processing and deploy it to a complex real-world many-core platform used for software development. In addition, we report achievable speedups and quantify the temporal error using a set of synthetic benchmarks.

8.5.1 Setup and Measurements

CoMix was tested using an integrated regressable test bench which starts and controls parallel execution of peers in individual shells. Peers run on different processors. Simulations were carried out on virtual machines using 2–4 host CPUs running CentOS as well as on a dedicated Intel-Xeon servers with four cores running RHEL.

We measure the application's *runtime* as the wall clock difference between end- and start-of-simulation callbacks. In distributed settings, we report the overall execution time as runtime of the slowest peer. *Speedups* are calculated dividing non-distributed by distributed runtime.

The *computation-to-runtime* ratio is calculated as the wall clock time a system spends between issuing transactions, divided by the overall runtime. For the synthetic benchmarks, this, e.g., is the time a producer spends in the loop body, outside of the (blocking) send-transaction call. As a more directly measurable variant, we also look at the number of transactions per second (wall clock) as an indication of the communication/computation ratio. The more computation a model performs per transaction, the fewer transactions are handled per second. In settings with constant total numbers of transactions, the throughput is also indicative of the overall runtime.

As an indication for the accumulated temporal error, we measure the overall SC time required for the execution of a particular software task (e.g., communicating a fixed number of tokens). The overall temporal error is calculated as the relative difference in SC time between distributed and non-distributed runs. A distributed simulation may require extra SC simulation time because initiators are waiting (i.e., are blocked) for responses from targets while their SC time advances.

8.5.2 Achievable Speedup

In order to quantify the overhead of our solution, we first look at a synthetic benchmark which combines a producer (P) and consumer (C) in one simulation part. The producer has a token-generating process that can be adjusted in its computational load (i.e., SC and host time consumption) per token, and issues a

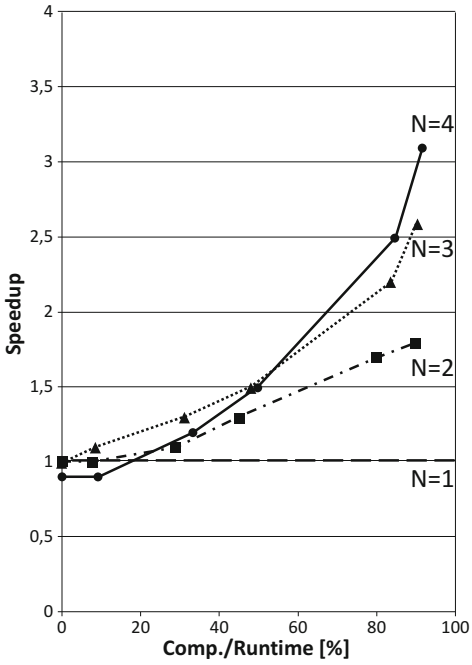


Fig. 8.5 Speedups for the synthetic benchmark distributed to $N = 2..4$ peers compared to the uncut simulation ($N = 1$)

token at the end of each iteration. The consumer receive tokens and verifies their sequence and inter arrival time without consuming SC time, it too can be adjusted in its computational load. Connectors are fully TLM compliant, which requires back signaling (cf. Sect. 8.4.2). Four of these parts are chained in a ring (P1-C2.P2-C3.P3-C4.P4-C1) and distributed onto up to four simulators, resulting in a symmetric load scenario. An additional parameter is the sync interval of the distributed parts which is kept constant for the measurement of the speedup (cf. Fig. 8.5). For the four simulator setting, for instance, a reasonable speedup of up to 3.2 is achieved depending on the computation to communication ratio. As expected, the figure confirms that no or only little speed can be gained for communication dominated settings (0–20 %).

8.5.3 Synchronization Interval

We look at a distributed producer-consumer benchmark (P1-C2) to analyze the sensitivity on the sync interval and report transactions/s in Fig. 8.6. Both parts have identical, generation-rate independent background task loads that are scheduled at 1/10th of the production interval. The throughput is impacted by the sync interval. Fine-grain synchronization limits the throughput, caused by the increased numbers of sync messages (also shown in the diagram) saturating the communication

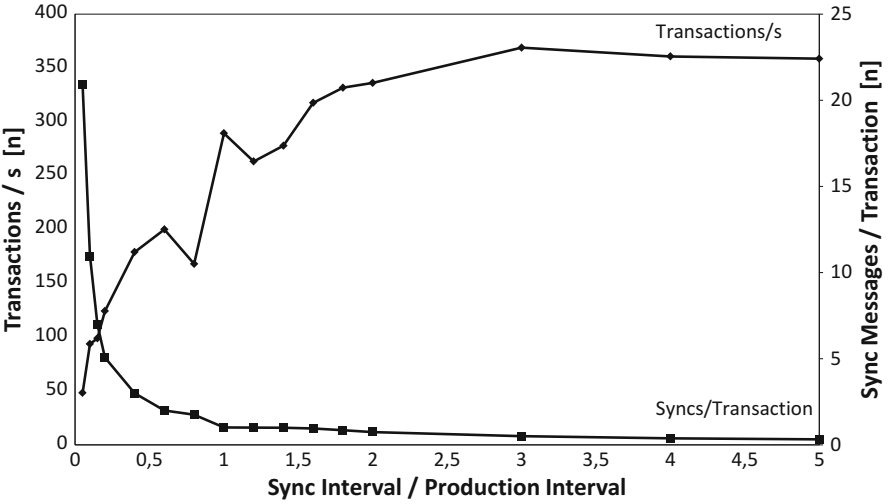


Fig. 8.6 Sensitivity of the simulation speed and throughput (transactions/s), on the sync interval (normalized to the production interval)

channel. For sync intervals set around the generation rate, some instability can be observed. Maximum throughput is reached for sync intervals set about 3X of the production interval. In this case, throughput is limited by the computational load of the background tasks, the latency of the communication channel, and by the temporal error. Larger sync intervals moderately increase temporal error further, leading to a slight degradation (cf. next section). For this measurement, connectors are used which do not synchronize transactions to the local time, as explained next.

8.5.4 Temporal Error

In settings with generation-rate independent computational tasks, the temporal error increases the overall runtime as these tasks continue to be executed, e.g., while the initiator is waiting for a response. But the computational background load also slows down the advancement of SC time which effectively lowers the temporal error up to a point where there is none. In contrast, an idle system, i.e., without background load, will always fast forward to the end of a synchronization interval, which means a response is never received before the end of the quota, so that the temporal error solely depends on the sync interval.

These effects can be modulated and (to some extent) compensated for by the schemes used for peer-to-peer and per-transaction synchronization, as Fig. 8.7 shows for the P1-C2 benchmark. With symmetric background task loads for producer and consumer (top), the temporal error has an upper bound that is relatively independent on the per-transaction synchronization scheme of the connector, fine-grain synchronization lowers the temporal error.

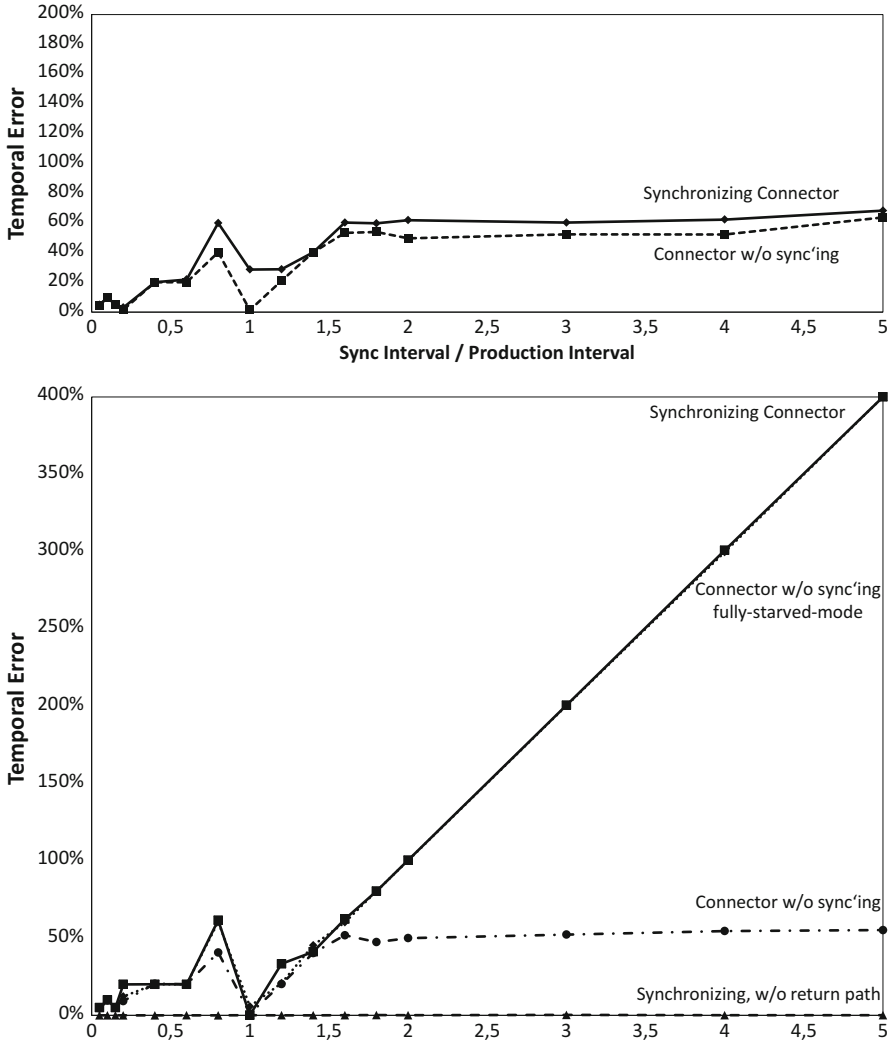


Fig. 8.7 Temporal errors for a symmetric load setting (*top*) and with idle consumer (*bottom*)

In cases of asymmetric loads (bottom, here with idle consumer), the smaller sync intervals cause the same temporal error as before. Above 1.6X, the temporal error increases linearly with the size of the sync interval for the synchronizing connector due to the end-of-quota effect, while the not-synchronizing connector remains at the constant level. However, this only is the case for the delta-only simulator synchronization, which handles late transactions still within the past quota. The fully-starved mode leads to the same error as the syncing connector. For our write-only setting, the temporal error is always negligible if the return path is avoided.

Table 8.2 Elaboration report for a real packet processing platform with four slices and some other functions resulting in about 5k instantiated SC modules

SystemC primitive	Slice	Platform
sc_modules	1037	4745
sc_ports	1459	6680
sc_signals	916	3962
sc_semaphores	8193	32772
sc_methods	436	2094
sc_threads	892	4410
sc_events	3645	17514
tlm2_initiator_sockets	1984	9457
tlm2_target_sockets	2280	10784

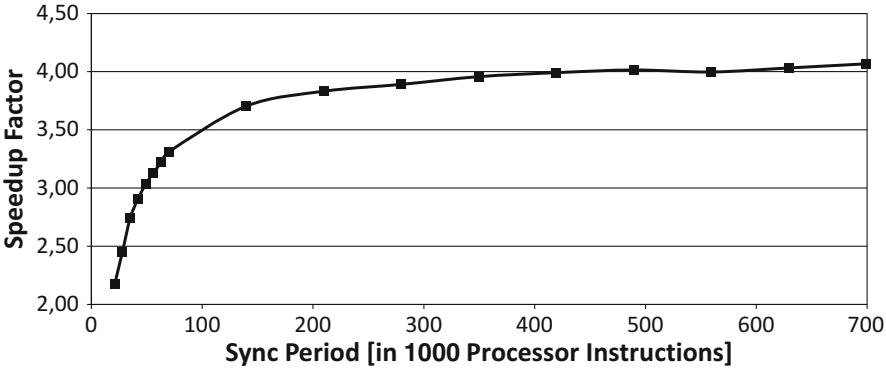


Fig. 8.8 Speedup for the real-world case study over the sync ratio

8.5.5 Packet Processing Platform

In a second step we apply CoMix to a real-world packet processing platform. The model has a considerable complexity as shown by its elaboration report in Table 8.2. For the purpose of this book chapter, we distribute a set of four slices into a simulation with four parts. Each of the parts comprises several 10s of binary-translating processor models that are busy running embedded software in temporally decoupled execution. The four parts are connected along write-only IO ports.

We vary the synchronization interval and run 10 simulations per data point to mitigate any load deviations on the simulation hosts. Figure 8.8 shows the speedup over the synchronization interval for the given setting expressed in processor instructions (CPI=1) and normalized to the clock frequency. Starting with about 380k instructions, a speedup of 4X is reached for the given setting, a computation dominated simulation setup with sparse communication and only loose synchronization between the parts.

8.6 Related Work

PDES Synchronization Policy Parallel discrete event simulation (PDES) is researched for several decades. SystemC is a discrete event simulator with unpredictable communication. According to Fujimoto [7] distributed SystemC simulation techniques can be categorized by their synchronization into conservative and optimistic approaches. Conservative schemes [4, 5, 12, 15] require the simulator to be aware of the minimum duration between two communication events in order to ensure temporal correctness while optimistic schemes [10] speculate on their future state. The former schemes impose high communication and synchronization overhead, especially with unpredictable communication (minimum sync period must be assumed), while the latter depend on checkpointing and rollback mechanisms in cases of incorrect speculation.

For unpredictable communication Peeters et al. [13] propose a hybrid synchronization scheme which (1) depends on write-exclusive access to shared memory for functional consistency, (2) avoids expensive frequent synchronization by accepting a temporal error in otherwise asynchronous communications, and (3) synchronizes explicitly at regular system-wide intervals using a blocking double handshake protocol. Similarly, our CoMix uses explicit synchronization intervals, but peers may grant different sync credits to each other which are received asynchronously and non-blocking. Sync messages must not be acknowledged explicitly. Shared memory and write-exclusive access is not required for functional consistency. Communication events from peers are received asynchronously but their processing is scheduled by the SC scheduler maintaining the single-threaded execution semantics of SystemC. CoMix customizable connectors support the full range of SC and TLM interfaces.

The conservative lookahead technique in [17] requires communication to be known ahead of time by at least on synchronization period, i.e., to be predictable. This avoids causality issues due to communication arriving late (as long as the return path is ignored [17]). In such a confined setting, CoMix does behave similarly accurate and without timing error (cf. Fig. 8.7).

SystemC Kernel Modifications Most prior approaches suggest changes to a the simulation kernel for adding communication, synchronization, or parallelization support, e.g., [4, 6, 10, 12, 15–17]. However, this causes a severe maintenance problem for evolving simulator versions and is not feasible in settings with heterogeneous, potentially commercial tools without source code access. Others avoid kernel modifications by providing add-on libraries which interact with the simulator only through the regular SystemC language interface [8, 13]. This interaction depends on the synchronization scheme and might be tight, e.g., per delta cycle as in [8], or rather loose. Our CoMix is such an overlay technology, interacting with the simulation engine only in cases of inbound communication events or explicit synchronization.

Peer-to-peer Protocols and Host Systems Several communication protocols are used for passing messages between peers, including MPI [5, 13], CORBA, and SOAP [11]. We use regular TCP/IP sockets similar to, e.g., [16] to limit the dependency on other libraries and a leaner protocol stack. Most related approaches use SMP machines as simulation hosts, e.g., [8, 10, 12, 15], potentially depending on SMP properties, such as shared memories and caches [13]. Our CoMix is intended for the use in load sharing facilities and geographically distributed settings, similar to [16]. However, CoMix recognizes the potential for optimizations and is factored for the support of other communication protocols.

SC/TLM Primitives and Modeling Styles Especially the kernel modifying approaches may impose special coding styles. Mello et al. [12], for instance, depend on approximately timed modeling semantics in their models. Others require thread safety, at least on distribution boundaries [15]. Both, kernel modifying approaches and overlay solutions often do not support the full spectrum of SystemC and TLM communication primitives [13, 17] or require explicit clocks [11]. Trams et al. [16] is limited to signal communication semantics. In [17], the partly supported TLM communication must not consume SC time. CoMix does not impose modeling restrictions and supports the full range of SC *and* TLM communication primitives by means of connectors for dedicated port/socket types.

8.7 Conclusion

We have presented CoMix, the Concurrent Model Interface, which enables the distributed simulation of large-scale SystemC-based virtual prototypes. CoMix provides robust communication between peers, enables their loose synchronization, and comprehensively manages the overall life cycle. Its modular design supports various synchronization strategies for peers and their communication, which may be chosen depending on a platform's specific requirements. CoMix' asynchronous IO infrastructure integrates into SystemC efficiently and avoid blocking third-party tools, such as embedded software debuggers.

For a set of synthetic, token-passing benchmarks we have shown the benefits of CoMix to be a trade-off between local computation and communication and the synchronization interval. The temporal error caused by the distribution can be lowered if double-synchronized round-trips are avoided by skipping the return path or not synchronizing it. These results were confirmed on a complex real-world platform, where we found speedups of up to 4X in a four part simulation for the given application. To date, CoMix is used in virtual prototypes of many-core network processing systems comprising several hundred instruction-precise processor models.

Acknowledgements In parts, this work has been supported by Lei Lang, Eric Frejd (Ericsson AB, Sweden), and Linmu Cui (Cadence, Germany).

References

1. Bailey, B., Martin, G.: Virtual prototypes and mixed abstraction modeling. In: ESL Models and their Application, pp. 173–224. Springer, Berlin (2010)
2. Bailey, B., McNamara, M., Balarin, F., Stellfox, M., Mosenson, G., Watanabe, Y.: TLM-Driven Design and Verification Methodology. Lulu Enterprises, Raleigh, NC (2010)
3. Benini, L., Flamand, E., Fuin, D., Melpignano, D.: "P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator," In: Design, Automation & Test in Europe Conference & Exhibition (DATE 2012), pp. 983–987, 12–16 March 2012. doi:10.1109/DATE.2012.6176639. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6176639&isnumber=6176405 (2012)
in Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 983–987, 12–16 (2012) doi: 10.1109/DATE.2012.6176639 http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6176639&isnumber=6176405
4. Combes, P., Caron, E., Desprez, F., Chopard, B., Zory, J.: Relaxing Synchronization in a parallel systemC kernel. International Symposium on Parallel and Distributed Processing with Applications (ISPA) (2008)
5. Cox, D.R.: RITSim: distributed systemC simulation. Master's thesis, Rochester Institute of Technology (2005)
6. Ezudheen, P., Chandran, P., Chandra, J., Simon, B., Ravi, D.: Parallelizing systemC kernel for fast hardware simulation on SMP machines. In: 23rd Workshop on Principles of Advanced and Distributed Simulation (PADS) (2009)
7. Fujimoto, R.M.: Parallel and distributed simulation. In: Proceedings of the Winter Simulation Conference (1999)
8. Huang, K., Bacivarov, I., Hugelshofer, F., Thiele, L.: Scalably distributed systemC simulation for embedded applications. In: International Symposium on Industrial Embedded Systems (SIES'08) (2008)
9. IEEE SystemC Language Reference Manual. IEEE Std 1666–2011 pp. 1–638 (2012)
10. Jones, S.: Optimistic parallelisation of systemC. Technical Report, University Joseph Fourier, MoSiG DEMIPS (2011)
11. Meftali, S., Dziri, A., Charest, L., Marquet, P., Dekeyser, J.L.: SOAP based distributed simulation environment for system-on-chip (SoC) design. In: Forum on Specification and Design Languages (FDL) (2005)
12. Mello, A., Maia, I., Greiner, A.; Pecheux, F.: "Parallel simulation of systemC TLM 2.0 compliant MPSoC on SMP workstations," In: Design, Automation & Test in Europe Conference & Exhibition (DATE 2010), pp. 606–609, 8–12 March 2010. doi:10.1109/DATE.2010.5457136. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5457136&isnumber=5456897 (2010)
13. Peeters, J., Ventroux, N., Sassolas, T., Lacassagne, L: "A systemc TLM framework for distributed simulation of complex systems with unpredictable communication," In: 2011 Conference on Design and Architectures for Signal and Image Processing (DASIP), pp. 1–8, 2–4 November 2011. doi:10.1109/DASIP.2011.6136847. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6136847&isnumber=6136840 (2011)
in Design and Architectures for Signal and Image Processing (DASIP), 2011 Conference on, pp. 1–8, 2–4 (2011) doi: 10.1109/DASIP.2011.6136847 http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6136847&isnumber=6136840
14. Sauer, C., Loeb, H.P.: A lightweight infrastructure for the dynamic creation and configuration of virtual platforms. In: 3rd Workshop on Virtual Prototyping of Parallel and Embedded Systems (VIPES) along with Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XV) (2015)
15. Schumacher, C., Leupers, R., Petras, D., Hoffmann, A: "parSC: synchronous parallel SystemC simulation on multi-core host architectures," In: 2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), pp. 241–246,

24–29 October 2010. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5751508&isnumber=5751486 (2010)
16. Trams, M.: Conservative distributed discrete event simulation with systemC using explicit lookahead. Technical Report, www.digital-force.net (2004)
17. Weinstock, J.H., Schumacher, C., Leupers, R., Ascheid, G., Tosoratto, L: "Time-decoupled parallel SystemC simulation," In: Design, Automation and Test in Europe Conference and Exhibition (DATE 2014), pp. 1–4, 24–28 March 2014. doi:10.7873/DATE.2014.204. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6800405&isnumber=6800201 (2014)

Part IV
Modelling and Verification of Power Properties

Chapter 9
Towards Satisfaction Checking of Power Contracts in Uppaal

Gregor Nitsche, Kim Grüttner, and Wolfgang Nebel

Abstract Since energy consumption is one of the most limiting factors for embedded and integrated systems, today's microelectronic design demands urgently for power-aware methodologies for early specification, design-space exploration, and verification of the designs' power properties. To this end, we currently develop a contract- and component-based design concept for power properties, called power contractss, to provide a formal link between the bottom-up power characterization of low-level system components and the top-down specification of the systems' high-level power intent. In this paper, we present a first proof of concept for the verification of the leaf-component power contracts of a hierarchical system design w.r.t. their implementation in UPPAAL. Building on these, we can provide assured power contracts for the hierarchical virtual integration (VI) of the leaf-components to a compound power contract of the integrated final system and thus allow for a sound and traceable bottom-up integration and verification methodology for power properties.

9.1 Introduction

Energy consumption has become one of the most limiting factors for today's embedded and integrated systems. As a consequence, the microelectronic design demands urgently for consistent methodologies which allow for an early specification, design-space exploration, and verification of the systems' power properties. To this end, different approaches are developed for high-level power estimation or an automatic synthesis, characterization, abstraction, and back-annotation of lower-level power characteristics. Nevertheless—being strongly dependent on future design decisions and low-level parameters, and since system and component power models can only

G. Nitsche (✉) • K. Grüttner
OFFIS—Institute for Information Technology, Oldenburg, Germany
e-mail: gregor.nitsche@offis.de; kim.gruettner@offis.de

W. Nebel
Carl von Ossietzky University Oldenburg, Oldenburg, Germany
e-mail: wolfgang.nebel@uni-oldenburg.de

© Springer International Publishing Switzerland 2016

F. Oppenheimer, J.L. Medina Pasaje (eds.), *Languages, Design Methods, and Tools for Electronic System Design*, Lecture Notes in Electrical Engineering 361,
DOI 10.1007/978-3-319-24457-0_9

provide a constrained validity, the reliability of such power estimations is strictly uncertain, depending heavily on the correct re-use of the power models within a proper *environment*.

To address this problem of *power closure*, we propose a contract- and component-based design concept, called power contracts [21], to provide a formal link between the top-down specification of the systems' high-level *power intent* and the bottom-up power characterization of low-level system components. For that purpose, we apply the ideas of *heterogeneous rich component (*HRC*)* [7, 8, 14, 22] and *contract based design (*CBD*)* [5, 8, 12, 22] to enable a component-based re-use of reliable power properties in a hierarchical design. Although we give an overview of our complete concept of extra-functional design with power contract, this paper will finally focus on the *leaf nodes* of the *virtual integration* process, meaning the specification, implementation, and verification of the contracts at the lowest abstraction level of the virtual integration process.

To introduce our extra-functional design concept with power contracts, we first give an overview of the underlying basic concepts and outline the complete methodology in Sect. 9.2. After that, we summarize the related work in Sect. 9.3 and give a more detailed understanding of power contracts in Sect. 9.4. To subsequently explain our methodology in detail, we show how we use contracts for the functional and extra-functional specification of system components in Sect. 9.5. In Sect. 9.6 we outline how we integrate the components' functional implementations with their extra-functional timing and power aspects—obtained from a bottom-up timing and power characterization—to derive an appropriate *multi aspect* verification model in *UPPAAL*. Concluding the explanation of our concepts, we give a short draft of our UPPAAL-based verification methodology in Sect. 9.7. As a first proof of concept we apply our methodology to the artificial example of an advanced encryption standard (*AES*) system, presented in Sect. 9.8. In Sect. 9.9 we finally summarize the results of our investigation and give an outlook to future work.

9.2 Basic Concept

An HRC denotes a structural design element—onwards component—which is semantically enriched with *contracts*, with contracts being a formal specification over the component's interfaces, declaring *assumptions* on the component's *environment* and *guarantees* on its externally observable behavior. Hence, the external interaction of an HRC is solely restricted to its explicitly declared interface. On top of that, its heterogeneity results from combining the behavioral descriptions of different, functional and extra-functional aspects within the same HRC.

To explain the most relevant concepts of HRCs and CBD, we introduce different identifiers according to Fig. 9.1, onwards denoting contracts by the letter C and HRCs by the letter M, additionally indexed by $i \in \mathbb{N}^+$ to refer to the ith HRC of a decomposition of M into n sub-HRCs $\{M_1, \ldots, M_n\}$, also denoted as *parts* of the system. Additionally, we define the interface of an HRC M as the set of its

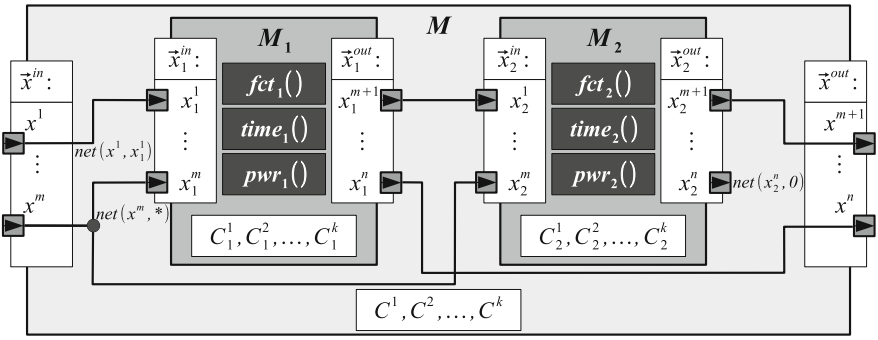


Fig. 9.1 Contract-Based Design (CBD): identifiers and basic concept

directed in- and output variables $x \in \chi_M := \{\mathbf{x}^{\text{in}}, \mathbf{x}^{\text{out}}\}$, called *ports*. To chose only the functional- , timing-, and power-specific subsets of M, C, χ, and $\mathbf{x}$, we provide the subscripts *fct, time*, and *pwr*, corresponding to the internal, non-structural but aspect-specific segregation of the HRC behavior according to these aspects. Finally, we define an HRC's interconnection network $\text{Net} := \{\text{Net}_{\text{asm}}, \text{Net}_{\text{del}}\}$ by the sets of its internal connectors, consisting of:

- the *assembly connectors* $\text{Net}_{\text{asm}} \subseteq \mathbf{x}^{\text{out}}_{M_i} \times \mathbf{x}^{\text{in}}_{M_j}; i, j \in \{1, \ldots, n\}; M_i, M_j \in M$ which internally link ports between different parts of the system;
- and its *delegation connectors* $\text{Net}_{\text{del}} \subseteq \{\mathbf{x}^{\text{out}}_{M_i} \times \mathbf{x}^{\text{out}}_{M}\} \cup \{\mathbf{x}^{\text{in}}_{M} \times \mathbf{x}^{\text{in}}_{M_i}\}; i \in \{1, \ldots, n\}; M_i \in M$ which link up the port of the HRC's parts with the HRC's external ports.

The direction of the interconnect $net(x_{\text{src}}, x_{\text{snk}}) \in \text{Net}$ is defined by position, naming the driving source x_{src} of the net in front of the reader's sink x_{snk}. Additionally, we summarize multiple connections from a common source by $net(x_{src}, *)$, respectively multiple connections to a common sink by $net(*, x_{snk})$ and denote open ports by $net(x_{src}, 0)$ or $net(0, x_{snk})$.

The *contracts* C of a component M are formally defined as triples $C := (A, B, G)$. While the

- *strong assumptions* A delimit the component's maximum permissible input state space over the input variables $\mathbf{x}^{\text{in}}$ of M,
- the *weak assumptions* B over $\mathbf{x}^{\text{in}}$ perform a further division to subspaces,
- for which M assures the *associated guarantees* G over its output variables $\mathbf{x}^{\text{out}}$, if and only if the individual use case satisfies the corresponding assumptions.

Hence, C is semantically interpreted as $[\![C]\!] := A \wedge B \Rightarrow G$, with A, B and G being time bounded LTL or CTL properties, representing sets of *timed traces* $S_A(\mathbf{x}^{\text{in}}), S_B(\mathbf{x}^{\text{in}})$ and $S_G(\mathbf{x}^{\text{out}})$ over the I/O variables $\mathbf{x}^{\text{in}}, \mathbf{x}^{\text{out}} \in \chi_M$ of M. Declaring the type of a variable x to be $\nu(x) \in \{\mathbb{B}, \mathbb{Z}, \mathbb{N}, \ldots\}$ and declaring the notion of time as the discrete but infinitely increasing variable $t \in \mathbb{N}_0^+$, a *timed trace* $s_x(t)$ is a discrete sequence of *events* $\{e(x, t_0), e(x, t_1), \ldots\} \in S_x := \{x \to [\mathbb{N}_0^+ \to \nu(x)]\}$, mapping the variable x to its *values* $v(x, t_i) \in \nu(x); i \in \mathbb{N}_0^+$ for each point of time.

A complete property specification C of a component M is then defined as $C := \bigwedge_{\text{asp}} \bigwedge_{i=0}^{n_{\text{asp}}} C_{\text{asp}}^{i}$, considering all contracts $\{C_{\text{asp}}^{1}, \ldots, C_{\text{asp}}^{n_{\text{asp}}}\}$ of all aspects $\text{asp} \in \{\text{fct}, \text{time}, \text{pwr}\}$.

To extract a purified, port-specific expression of the effect of the assumptions, guarantees or even complete contracts, the *restriction function* $\downarrow_X$ denotes the restriction of these constraints to solely the subset X of their original variables. Furthermore, ρ_i and ρ denote the so-called *port mapping* or port substitution functions, with: ρ_i identifies the port variables $x_i^{\text{in}}, x_i^{\text{out}} \in \chi_{M_i}$ of a part M_i with the corresponding assembly and delegation connectors $net(x_i^{in}, *), net(*, x_i^{out}) \in Net$; and ρ identifies the external ports $x^{\text{in}}, x^{\text{out}} \in \chi_M$ of the system M with the corresponding *delegation connectors* $net(x^{in}, *), net(*, x^{out}) \in Net$.

That way, the contracts explicitly relate the formal and possibly more abstract behavioral specifications of a component's bottom-up characterization with the appropriate validity constraints, the underlying model implementations and abstractions would otherwise assume to be satisfied without verification. Hence, CBD enables to formally check for:

- *compatibility*: $G_{M_{\text{src}}} \downarrow_{x_{\text{src}}} \rho_{M_{\text{src}}} \Rightarrow A_{M_{\text{snk}}} \downarrow_{x_{\text{snk}}} \rho_{M_{\text{snk}}}$ between the connected components of a system;
- *refinement*: $C' \Rightarrow C$ of a system M's specification C w.r.t. its component-based bottom-up composition by n parts $\{M_1, \ldots, M_n\}$, specified by their contracts C_i and logically composed to the *VI* $C' := \left(\left(\bigwedge_{i=1}^{n} C_i \rho_i\right) \rho \downarrow_{\chi_M}\right)$.

Applying the concepts of HRCs and CBD to build a consistent, power-aware design flow, our primary goal is to formally ensure the correct re-use of bottom-up leaf-node power models to improve power closure. Our basic idea for that design and verification flow is outlined in Fig. 9.2, covering:

1. the structural decomposition of the initial HRC with possibly a refined partitioning of its initial contracts;
2. the implementation of the HRC's parts;
3. the formal bottom-up characterization of the parts' functional and extra-functional behavior in terms of contracts;
4. the *satisfaction* checking between the parts' contract-based bottom-up characterization and their specification;
5. the compatibility checking between all components' connected ports;
6. the virtual integration to a composed top-level specification;
7. the refinement checking between the composed top level contract and those of the initial specification.

According to our focus on the verification of a hierarchical system's leaf-node power contracts versus the system's implementation in UPPAAL, in this paper we onwards consider only the lowest steps 2–4 of Fig. 9.2 in more detail. Building on this, we can provide assured power contracts for checking the compatibility and refinement of the hierarchical virtual integration of the leaf-components w.r.t. the integrated final system, allowing for a sound and traceable bottom-up integration and verification methodology for power properties. As a first proof of concept, we

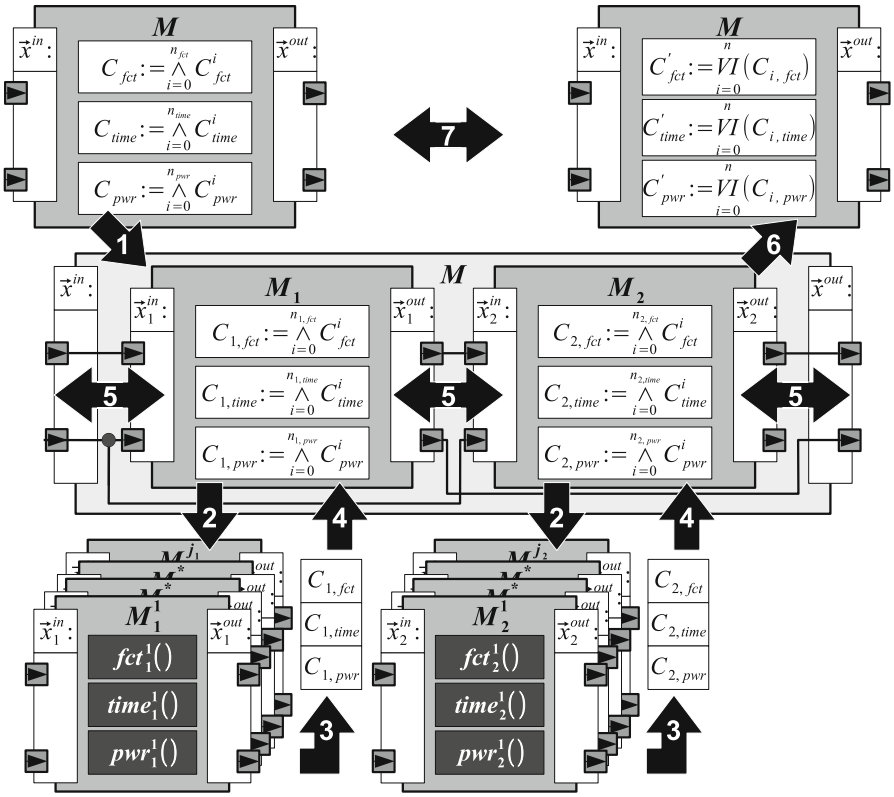


Fig. 9.2 Basic idea of the design steps within our power-aware design flow with power contracts

implemented our exemplary power contracts by queries and observer automata in UPPAAL [2, 15, 24] , based on the theories of communicating *timed automata* (*TA*) [1, 4, 9] and the timed execution traces of their variables.

9.3 Related Work

The objective of *Power Closure* within a consistent, power-aware design flow for embedded and integrated systems is part of several different approaches of research, of which the currently most related ones are: [3, 6, 10]. Nevertheless, to the authors best knowledge, there is no work addressing power closure with combining the contract- and component-based virtual integration design flow at the higher levels of abstraction with the low-level implementation and characterization flow, applying a multi aspect modeling and verification approach for formal satisfaction checking.

Differently to the distinction of [10] between approaches for power characterization and modeling and approaches for the analysis and verification of power management, the most recent approach of [6] addresses both, presenting a framework for the statistical modeling and analysis of schedulability and energy efficiency of embedded hierarchical scheduling systems running on multi-core platforms. For that purpose, tasks are modeled as UPPAAL *stop watch automata* (*SWA*) with additional parameters for the processor specific worst and best case execution times and power consumption rates. That way, the approach allows for analyzing and verifying the schedulability of tasks w.r.t. its impact on energy efficiency.

Likewise, [10] itself presents an approach addressing power and power management, as well. In detail, a multi-view system model is suggested to link the specialized models and tools of different domains according to a *UML* profile, based on *MARTE* and *SysML*. Similarly to our approach, functionality, timing impacts of the clock and power impacts of V_{DD} and power management are described in different views, but differently, these views use different models and tools, necessitating a valid transformation and synchronization to enable a combined analysis and verification.

Similarly, in [3] the multi-view UML profile *DIPLODOCUS* is extended to power and power management, focusing on CPUs, described by UML *Power State Machines* (*PSMs*) and UML power managers. To this aim, UML state machines are extended with *power states*, providing voltage, frequency and static power information, and with power transitions, annotated with the duration and possible power overheads of the transitions. Furthermore, the approach provides own tools for modeling and simulation, providing also an interface for formal verification with UPPAAL. Differently to this, we aim towards a contract and component-based design approach, which focuses on contracts and virtual integration as far as possible, interfacing the implementation oriented *model based design* (*MBD*) at lower levels of abstraction only for the characterization and verification of the leaf-nodes.

9.4 Power Contracts

To establish the formal link between the high-level power requirements of a system—called *power intent*—and its low-level implementation resp. composition of system components, we propose power contracts [21] as a CBD technique, which allows to formally specify and verify the constraints for a correct re-use of power models w.r.t. their application and abstraction in a compositional design.

Addressing the most relevant factors of *dynamic power* consumption, our current notion of power contracts is as given in Fig. 9.3. Integrating or linking the power contract within an HRC by the means of its HRC reference, the contracts' constraints become a traceable and verifiable property description of that component. Within their *strong assumptions* the different implementations of an HRC are distinguished w.r.t. the applied *technology*, *architecture*, and *power domain*—each

Power Contract		
HRC Reference		
A:	Implementation	Technology, Architecture, Power Domain
B:	Functional Mode	I/O Values
	Power Mode	Voltage, Frequency
G:	Power Consumption	State Power, Power Gradient, Minimum Power, Maximum Power, Average Power

Fig. 9.3 Content and structure of a power contract

defining an unambiguous reference to exactly: one specific *design kit* version of the technology; one specific structural implementation of a *design library*, called architecture; or one specific power domain definition.

The latter might partially be obtained from a component's IEEE 1801 Unified Power Format or Common Power Format specification [13, 23], but, as we denote the *power modes* as combined voltage–frequency tuples $\overrightarrow{pm}^{in} := (V_{\mathrm{DD}}, f_{clk})$, additional frequency information must be provided to entirely declare the admissible power modes of a power domain. Since technology, architecture and power domain constraints are independent from time, it is sufficient to independently check satisfaction or violation of those parameters before verifying the more complex, dynamic state space of the timed I/O variables.

Subsequently, the weak assumptions of a power contract restrict the validity of its guarantees w.r.t. to the dynamics of the input variables $\mathbf{x}^{in}_{\mathrm{pwr}} := (\mathbf{x}^{in}_{\mathrm{fct}}, \overrightarrow{pm}^{in})$, comprised of the *functional mode*, given by the functional inputs $\mathbf{x}^{\mathrm{in}}_{\mathrm{fct}}$, and the *power mode*, $\overrightarrow{pm}^{in}$ defined by the timed traces of the supply voltage V_{DD} and the operating frequency f_{clk}.

Hence, if the chosen implementation of the HRC satisfies the *strong assumptions* and if the embedding environment satisfies the *weak assumptions*, a power contract provides an assured guarantee w.r.t. the HRC's power consumption, which is formally protected against faulty re-use. Denoting the time by t or $t_{[]} := [t_l, t_u]$ for time intervals, $t, t_l, t_u \in \mathbb{N}_0^+$; the current power contracts provide the following specification concepts for those guarantees:

- the current state power consumption at time t:

$$p(\mathbf{x}^{\mathrm{in}}_{\mathrm{pwr}}, t) := 1/2 \cdot C \cdot \bar{\alpha}(\mathbf{x}^{\mathrm{in}}_{\mathrm{fct}}, t) \cdot V^2_{\mathrm{DD}}(t) \cdot f_{\mathrm{clk}}(t);$$

- the average power gradient at time t_u resp. during $t_{[]}$:

$$p_{\mathrm{grd}}(\mathbf{x}^{\mathrm{in}}_{\mathrm{pwr}}, t_{[]}) := 1/\Delta t \cdot (p(\mathbf{x}^{\mathrm{in}}_{\mathrm{pwr}}, t_u) - p(\mathbf{x}^{\mathrm{in}}_{\mathrm{pwr}}, t_l));$$

$$p_{\mathrm{grd}}(\mathbf{x}^{\mathrm{in}}_{\mathrm{pwr}}, t_u) := p_{\mathrm{grd}}(\mathbf{x}^{\mathrm{in}}_{\mathrm{pwr}}, t_{u-1}, t_u);$$

- the minimum or maximum power consumption during $t_{[]}$:

$$p_{\min}(\mathbf{x}^{\text{in}}_{\text{pwr}}, t_{[]}) := \min_{t \in t_{[]}}(p(\mathbf{x}^{\text{in}}_{\text{pwr}}, t));$$

$$p_{\max}(\mathbf{x}^{\text{in}}_{\text{pwr}}, t_{[]}) := \max_{t \in t_{[]}}(p(\mathbf{x}^{\text{in}}_{\text{pwr}}, t));$$

- the average power consumption during $t_{[]}$:

$$p_{\text{avg}}(\mathbf{x}^{\text{in}}_{\text{pwr}}, t_{[]}) := \frac{1}{\Delta t} \cdot \sum_{t_i = t_l}^{t_u - 1} p(\mathbf{x}^{\text{in}}_{\text{pwr}}, t_i) \cdot (t_{i+1} - t_i).$$

At that, we currently consider only dynamic power consumption according to our power characterization by average *switched capacitances* $\bar{C}_{\text{sw}} := 1/2 \cdot C \cdot \bar{\alpha}(\mathbf{x}^{\text{in}}_{\text{fct}}, t)$ [19, 20], with the overall circuit's switched capacitance C and the proportionality factor $\bar{\alpha}(\mathbf{x}^{\text{in}}_{\text{fct}}, t)$; $0 \leq \bar{\alpha} \leq 1$; denoting the fractional amount of C that is switching at time t according to the functional inputs $\mathbf{x}^{\text{in}}_{\text{fct}}$. This is not a general limitation of our concept. Quite contrary, the methodology would similarly fit for *static power* consumption, extending the assumptions of the power contracts to also consider the temperature and basing the power characterization on *leaking resistance*.

9.5 Leaf-Node Specification with Power Contracts

As the origin of our leaf-node implementation and verification cycle—i.e., steps 2–4 in Fig. 9.2—the CBD and VI flow ends with aspect-specific contracts as the final top-down specification of components at its lowest levels of abstraction. To follow the paradigms of CBD and *separation of concerns* we base our specifications on solely an exterior view of the components' ports, using the following denotational semantics, inspired by *tagged signals* [16–18]:

Starting with the initially completely *untimed*, functional aspect of the specification we denote each *untimed trace* as a function $s_x(i) := \{e_0(x), e_1(x), \ldots e_n(x)\}$ mapping the port x to a totally ordered sequence of i-indexed events $e_i(x) = e(x, i) := (v_i(x), i)$, resp. to its i-th values $v_i(x) = v(x, i) \in v(x)$, using continuously increasing indices $i \in 0, 1, \ldots \subseteq \mathbb{N}_0^+$.

As the total order $(i, \leq)$ of this pure functional perspective determines only the sequentiality among the events of a single trace but not w.r.t. the events of different traces, pure functional specifications cannot define the *causality* between events of different ports. If this most abstract notion of timing is desired within the functional domain, a common, port-independent index $i' \in \mathbb{N}_0^+$ becomes necessary, to define their partial order $(\mathbb{N}_0^+, \leq)$ according to a mapping to i'. Hence, a causality specification for the events of different ports becomes possible by constraining that mapping from the originally i-indexed traces $s_x(i)$ to the i'-indexed event sequence

$s_{\mathbf{x}}(i')$, with $s_{\mathbf{x}}(i')$ allowing for simultaneity of events $e(x_1, i'_1), \ldots, e(x_n, i'_n)$, indexing them with the same index $i'_1 = \cdots = i'_n$. With the ordering operators $\sqsubset, \sqsubseteq$ a specification of causality becomes expressible, using $e(x_1, i_1) \sqsubset e(x_2, i_2)$ to demand the event $e(x_1, i_1)$ to really precede $e(x_2, i_2)$, i.e., $i'_1 < i'_2$; resp. $e(x_1, i_1) \sqsubseteq e(x_2, i_2)$ to also allow simultaneity, i.e., $i'_1 \leq i'_2$.

For the more detailed specification of timings, the timing view enforces this mapping for all traces of the system, using a common notion of time $t \in \mathbb{R}_0^+$ as the additional index, simplifying to $t \in \mathbb{N}_0^+$ for discrete time and clocked systems. Based on that, we denote the resulting *timed traces* as functions $s_x(t) := \{e_0(x), e_1(x), \ldots e_n(x)\}$ which again map a port x to a totally ordered sequence of i-indexed events, but with $e_i(x) = e(x, t_i) := (v_i(x), t_i(x))$ denoting its values $v_i(x) = v(x, t_i) \in \nu(x)$ for each evaluation point of time $t_i(x)$. Hence, while the previous indices i' enable only a partial order $(\mathbb{N}_0^+, \leq)$, i.e., $(i'_1 \leq i'_2) \in 0, 1$, the physical interpretation of t in the metric space $(\mathbb{N}_0^+, |t_i, t_j|)$ allows for a metric specification of "absolute" points of time t_i—that means "relative" w.r.t. the initial point of time $t_0 = 0$—resp. time differences $t_i - t_j$, representing state and process durations, time ranges, and delays of events. Since these timings most often are frequency dependent for clocked systems, our timing view extends the components by the extra-functional *frequency port* f_{clk} to allow for timing specifications, relative to the clock cycle $T_{\text{clk}} = 1/f_{\text{clk}}$.

To finally enable power contracts to specify the components' power consumption, we extend the components to the power aspect, adding the ports for the voltage supply V_{DD} and the *power port* p of the power consumption. Hence, based on the incremental extension of the components' functional aspects by timing and power, *power contracts* are supposed to comprehensively specify the components' power behavior w.r.t. the influences from all of these aspects. A more detailed explanation of this is given by the example in Sect. 9.8.1.

9.6 Leaf-Node Implementation in UPPAAL

Similarly to the incremental specification process, the implementation of the components' verification models in UPPAAL is incrementally derived from first the functional, then the extra-functional specifications, too. This can be done by the re-use of pre-characterized library components, which themselves provide already verified functional and extra-functional contracts to enable bottom-up virtual integration according to steps 4–7 of Fig. 9.2. If no fitting HRCs are available, steps 2–4 have to be passed to derive such an implementation, which satisfies the functional contract, and to characterize and verify its extra-functional behavior w.r.t. to timing and power. To link these steps of the low-level design process with the contract-based high-level design, we use the following implementation models for the functional, timing, and power aspects.

To finally derive a complete multi aspect HRC, which is verifiable using UPPAAL, an appropriate integration of the components' functional, timing, and

power consumption as a network of TAs becomes necessary. Since the separation of concerns on the other hand demands for an independent description of these—anyway differently obtained—extra-functional characteristics we currently use a loosely interweaving, which methodically extends the functional model by the extra-function information.

Initially we realize ports $x \in \chi$ (resp. their traces) via tuples $(v(x), e(x))$ of a global integer variable v combined with an associated broadcast channel e. As the indices of a trace-based specification actually describe history—i.e., memory behavior—and as the interconnect nets are supposed to be memory-less, the indices are no explicit part of the ports implementation. As a consequence, indices have to be either part of the components, actually implementing real memory, or as a part of the verification environment, observing the trace for the specific amount of values.

9.6.1 Functionality and Causality

Starting with the functional contracts a UPPAAL network of TA without clocks can be derived, to describe a component's functional implementation M_{fct} by *communicating extended finite state machines* [11].

Definition 1 (Communicating Extended Finite State Machine (CEFSM)). The CEFSM is defined as a tuple $CEFSM := (Q, q_0, \text{Edg}, \text{Var}, \chi, E_\chi, \text{Act}, \text{Inv});$, with

- states $q \in Q$,
- initial state $q_0 \in Q$,
- edges $\text{edg} \in \text{Edg} \subseteq Q \times \text{Act} \times Q$,
- local variables $\text{var} \in \text{Var}$,
- port variables $x \in \chi$,
- associated communication channels $e_x \in E_\chi$,
- actions labels $\text{act} \in \text{Act} = \{E?, E!, \varepsilon\} \times G \times U$ and
- *invariants* $\text{inv} \in \text{Inv} : Q \to f_{\mathbb{B}}(\text{Var}, \chi)$.

Actions are defined in the following:

- $g \in G : \text{Edg} \to f_{\mathbb{B}}(\text{Var}, \chi)$
  denotes the transitions' *guard*s, enabling the transition w.r.t. to the Boolean result of the relational, arithmetic, and Boolean operation $f_{\mathbb{B}}(\text{Var}, \chi)$;
- $u \in U : \text{Edg} \times \text{Var} \times \chi \to \nu(\text{Var}) \times \nu(\chi)$
  denotes the transitions' variable *updates*, assigning them new values according to their variable type ν;
- $e? \in E? : E_\chi \to \mathbb{B}$ and $e! \in E! : E_\chi \to \mathbb{B}$
  describe a transition's destructive read resp. write event access on the communication channels, synchronously triggering the receiving transition with executing the sending transition; and
- ε denotes the internal null action, enabling a transition for spontaneous execution. □

Alternative to the manual top-down design of a component's functional high-level model, the *communicating extended finite state machine* of its component's functional behavior can be obtained by the bottom-up formalization and abstraction of an available RT or gate-level implementation, e. g. using *sound abstractions* [25].

To integrate *untimed causality specifications* we extend our notion of ports and traces by an additional integer semaphore $lck(x)$, which is incremented by each component receiving $e(x)$ via $e(x)?$, and which blocks the event's source component from proceeding execution until all receivers committed the completion of their untimed, atomic reactions on the event, decrementing $lck(x)$ back to null. For that, we use $\oplus x$ to abbreviate the semaphore's increment $++lck(x)$ at the receiving edges $e(x)?$ resp. $\ominus x$ as the semaphore's decrement $--lck(x)$ at an arbitrary transition of that components' implementations, which have a non-interruptable reaction on the event.

9.6.2 Timing

The implementation of the more detailed timing aspect is obtained by extending the extended state space $\bigcup_M(Q \times \text{Var} \times \chi)$ of the CEFSM network to $\bigcup_M(Q \times \text{Var} \times \chi) \times \text{clk}$, extending the CEFSM with clocks, called *communicating extended timed automata*.

Definition 2 (Communicating Extended Timed Automata (CETA)). The CETA is defined as a tuple $CETA := (Q, q_0, \text{Edg}, \text{Clk}, \text{Var}, \chi, E_\chi, \text{Act}, \text{Inv})$, which is defined like a CEFSM, with the following extensions:

- $\text{clk} \in \text{Clks}$ as a metric of time $t(\text{clk}) : \text{clk} \to \mathbb{R}_0^+$,
- $\text{inv} \in \text{Inv} : Q \to f_{\mathbb{B}}(\text{Var}, \chi, \text{Clks})$,
- $g \in G : \text{Edg} \to f_{\mathbb{B}}(\text{Var}, \chi, \text{Clks})$ and
- $u \in U : \text{Edg} \times \text{Var} \times \chi \to \nu(\text{Var}) \times \nu(\chi) \times 0, t(\text{clk})^{|\text{Clks}|}$. □

Hence, a system's timing implementation becomes a network of CETA.

To interweave the functional implementation with the top-down timing specification resp. with a bottom-up timing characterization, we first introduce a global clock clk^* as a common time reference $t = t(\text{clk}^*)$ within the TA network plus a necessary number of additional local clocks $\text{clk}_{\text{edg}} \in \text{Clks}$ per component, to allow for component specific perceptions of time $t(\text{clk})$.

Furthermore, the discretely timed, i.e., clocked component obtain a frequency port f_{clk}, as an interface to receive external time step information, which locally maybe different from other components.

On this basis, each edge $\text{edg} \in \text{Edg}_{CEFSM}$ between the functional model's originally committed states $q \in Q_{CEFSM}$—i.e., *untimed states*, which allow no progress of time—is replaced by a sequence of edges $\text{edg}_{t,1}, \ldots, \text{edg}_{t,n} \in \text{Edg}_t \subset \text{Edg}_{CETA}$ and timed states $q_{t,1}, \ldots, q_{t,n-1} \in Q_t \subset Q_{CETA}$ which may be traversed according to their guards $g(\text{edg}_t)$ and invariants $\text{inv}(q_t)$, allowing time to advance within the constraints of the invariants. Hence, if the edge's source state q is no

subject of a timing specification, it is sufficient to just replace that state with an equivalent timed state. Otherwise, to implement the timing constraints of the other states, a clock $\mathrm{clk}_{\mathrm{edg}} \in \mathrm{Clks}$ is dedicated to that sequence, being reset by the means of $u(\mathrm{edg}_{t,1}) : t(\mathrm{clk}_{\mathrm{edg}}) = 0$; at the initial transition $e_{t,1}$ and measuring the progress of time along the further transitions of that sequence.

Additionally, the initial transition inherits the functional guard $g(\mathrm{edg})$ and—if not constraint by the further specifications of causality and timing—the first updates $u(\mathrm{edg})$ of the former edge. The additional timed states q_t are then annotated with time invariants $\mathrm{inv}(q_t) \in \{t(\mathrm{clk}_{\mathrm{edg}}) \bowtie \mathbb{N}_0^+\}; \bowtie \in \{<, \leq\}$; which delimit the maximum progress of time, allowed for that state according to the specification of event delays and process durations.

Similarly, complementing the invariant $\mathrm{inv}(q_{t,i-1})$ of a source state according to the timing specifications, the outgoing edges $\mathrm{edg}_{t,i}$ of each added state $q_{t,i-1}$ are annotated by further guards $t(\mathrm{clk}_{edg}) \bowtie \mathbb{N}_0^+; \bowtie \in \{<, >, \leq, \geq\}$ to disable this edge until at least the specified minimum of time has passed.

According to this, if a functional edge triggers multiple updates with different specifications of timing, for each update an additional pair of a timed state and an edge with this update is appropriately appended according to the sequential update order at the original edge. At that, the affected channel synchronizations $e(x)!$ of the original edges are applied to the edges $\mathrm{edg}_{t,i}$ with the corresponding writing update $u(\mathrm{edg}_{t,i} : x \rightarrow \nu(x)$, writing to the corresponding port x, resp. $e(x)?$ to the first transition $\mathrm{edg}_{t,1}$ of the substituting timed transition sequence. As the timing should refine the semaphore synchronizations of the causality-extended functional implementation, possibly affected $\oplus x$ and $\ominus x$ operations of the edges can be omitted.

Besides the manual top-down specification of timings a component's bottom-up timing characterization can be obtained, applying standard methods of timing analysis, as e. g. *static timing analysis* (*STA*) and *control data flow graph* (*CDFG*) scheduling.

9.6.3 Power

Finally, when the functional and timed aspects are interwoven, the component's power aspect has to be integrated. Hence, under the assumption of discretely timed changes of the power consumption, we provide the following extensions to the previously described networks of CETA to obtain *communicating extended timed automata with power* (*CETA+p*).

Definition 3 (Communicating Extended Timed Automata with Power). A *CETA+p* is a CETA with the following extensions:

- A switched capacitance variable $\bar{C}_{\mathrm{sw}} \in \mathrm{Var}$ is added to the component's local variables Var.

- All edges of the CETA are annotated with update functions $\{u_{\bar{C}_{\mathrm{sw}}}\}(\mathrm{edg}_{t,i})$: $\mathrm{edg}_{t,i} \rightarrow \nu(\bar{C}_{\mathrm{sw}}(q_{t,i}))$ to relate their target—onwards denoted as *power states* $q_{\mathrm{ps}} \in Q_t$—with the corresponding switched capacitance.
- The components' port interface χ is extended by the supply voltage input port V_{DD} and the power port p, always directed as output. □

As the power port is virtual, meaning it is no structural port of the design, and according to its dedicated physical interpretation as the component's power consumption, the interconnect of power ports is implicitly defined as a delegation connector relating a system's power port p_M with all power ports of its parts p_{M_i}. Differently to connectors with structurally existing implementations the virtual connector of the power ports always summarizes the current power values of all parts' power ports, i.e., $v(p_M, t) := \sum_{M_i} v(p_{M_i}, t)$; whenever an update event occurs at one of them, delivering the result to the top-level power port of the system.

Combined with the previously explained power evaluation and abstraction functions $p(\mathbf{x}^{\mathrm{in}}_{\mathrm{pwr}}, p_{\mathrm{grd}}(\mathbf{x}^{\mathrm{in}}_{\mathrm{pwr}}, t_{[]}), p_{\mathrm{min}}(\mathbf{x}^{\mathrm{in}}_{\mathrm{pwr}}, t_{[]}), p_{\mathrm{max}}(\mathbf{x}^{\mathrm{in}}_{\mathrm{pwr}}, t_{[]}),$ and $p_{\mathrm{avg}}(\mathbf{x}^{\mathrm{in}}_{\mathrm{pwr}}, t_{[]})$ this provides our basic framework for the implementation of the power aspect.

To achieve the previously described switched capacitance mapping we use a bottom-up characterization by fine-granular *Protocol State Machines* and *Power State Machines* (*PrSM/PSM*) [19, 20], obtained according to Fig. 9.4. That way, the components' dynamic power consumptions are derived by correlating their observable communication at the functional ports $\mathbf{x}^{\mathrm{in}}_{\mathrm{fct}}$ with the according power-over-time trace $p(\mathbf{x}^{\mathrm{in}}_{\mathrm{pwr}}, t)$, obtained from, e. g. a detailed gate-level power estimation. The resulting Protocol State Machine then represents the components' timed protocol implementation, relating the observable functional I/O communication to timed events, which describe the components' internal functional state $q \in Q$, consequently influencing its power consumption.

For simplification, we use the most detailed switched capacitance mapping, characterizing all of a component's timed states as *power states* $Q_{\mathrm{ps}} = Q_t \subseteq Q_{CETA}$, assuming it as a *white-box* model. As white-box models, openly provide detailed knowledge of the component's state transition and output functions—contained within the edges $\mathrm{Edg} \subseteq Q \times \mathrm{Act} \times Q$ and action labels $\mathrm{Act} = \{E?, E!, \varepsilon\} \times G \times U$—the power characterization may become complete and time accurate, by triggering each of the component's internal states q by the appropriate input stimuli.

Differently, for the case of an abstract or *black-box* model characterization the approach allows for arbitrary abstractions $q_{\mathrm{ps}} \in Q_{\mathrm{ps}} \subseteq Q$ of the internal states $q \in Q$, based on some correlation of the timed I/O traces $\mathbf{x}^{\mathrm{in}}_{\mathrm{fct}}, \mathbf{x}^{\mathrm{out}}_{\mathrm{fct}}$, with the associated power trace $p(\mathbf{x}^{\mathrm{in}}_{\mathrm{pwr}}, t)$. To relate the I/O traces with the power consumption, the PrSM controls a corresponding PSM, which provides the average *switched capacitances* $\bar{C}_{\mathrm{sw}}(q_{\mathrm{ps}}) := 1/2 \cdot \hat{C}_{\mathrm{sw}} \cdot \bar{\alpha}(q_{\mathrm{ps}})$ according to each of the PrSM's power states q_{ps}, with $\hat{C}_{\mathrm{sw}}$ being the implemented circuit's total switched capacitance and $\bar{\alpha}(q) \in \mathbb{R}; 0 \leq \bar{\alpha}(q) \leq 1$ denoting its fractional switching activity in the state q. Combined with the voltage and frequency values $(V_{\mathrm{DD,est}}, f_{\mathrm{clk,est}})$, used during the power estimation, the average switched capacitance of a power state q_{ps} can be derived according to:

$$\bar{C}_{sw}(q_{ps}) = \frac{p(\mathbf{x}^{in}_{\text{pwr}},t)}{(V^2_{\text{DD},est} \cdot f_{clk,est})}$$

$$\text{for:} \quad \mathbf{x}^{in}_{\text{fct}} = q_{ps}; \qquad \overrightarrow{pm} = (V_{\text{DD},est}, f_{clk,est});$$

$$\text{and:} \quad t = t(e(q,i)); \qquad q = q_{ps};$$

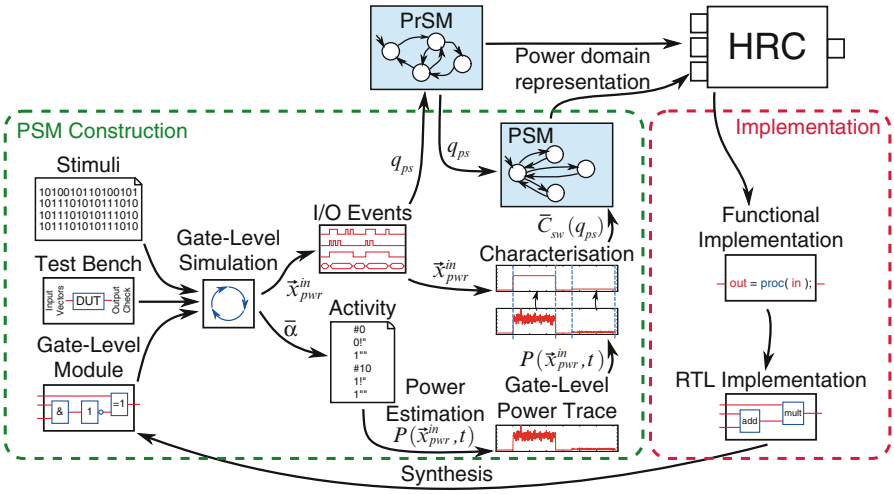


Fig. 9.4 Characterization of the HRCs' power behavior using the PrSM and PSM approach

9.7 Observer Implementation and Verification in UPPAAL

In the end, the bottom-up implementation and characterization can be checked against the contract-based specification, to verify the satisfaction of the contracts. For that purpose, the comprised multi aspect HRC is embedded within an environment of timed automata which drive the input ports of the system. Furthermore, the contracts of the specification are implemented as a combination of properties which are expressed in the *UPPAAL Requirements Specification Language* (*UPPAAL-RSL*) [2, 24] and additional observer automata, comparing all possible traces of the implementation with the traces of the specification, returning a *sat* trace, which evaluates to $v(sat,t) = 1$ if the property is satisfied at the present evaluation point t of time, resp. to $v(sat,t) = 0$ if the property is violated. Hence, to apply an observer $\text{OBS}_{M_i,asp}$ we formally check the UPPAAL-RSL property $A\Box(\text{OBS}_{M_i,asp}.sat \wedge !deadlock)$.

$$C^1_{\text{AES,fct}}\,(\mathbf{x}^{in}_{\text{AES,fct}} = in;\, \mathbf{x}^{out}_{\text{AES,fct}} = out;\, \nu(in) : \text{uint8};\, \nu(out) : \text{uint24};) :$$
$$A^1_{\text{AES,fct}} : true;$$
$$B^1_{\text{AES,fct}} : e^{in}_{3i-2} \wedge e^{in}_{3i-1} \wedge e^{in}_{3i} \wedge v^{in}_i \in [0,255] \wedge i \in \mathbb{N}^+;$$
$$G^1_{\text{AES,fct}} : e^{out}_i \wedge v^{out}_i = f_{\text{AES}}(v^{in}_{3i}, v^{in}_{3i-1}, v^{in}_{3i-2});$$

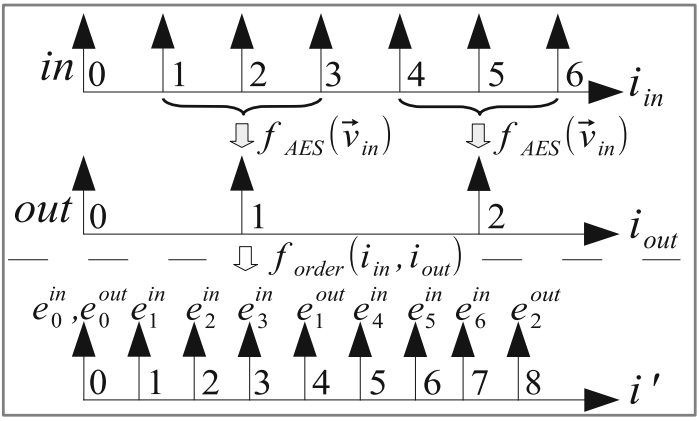


Fig. 9.5 AES functional specification via $C^1_{\text{AES, fct}}$

9.8 Proof of Concept

For a first evaluation of power contracts, we investigated the simple AES (*advanced encryption standard*) example from [21] in UPPAAL, implementing the leaf-node HRCs according to Sect. 9.6 as a network of timed automata and using the (UPPAAL-RSL) [2, 24] combined with additional *observer automata* for the specification and verification of our contracts.

9.8.1 AES Specification

Starting with an interface definition $\chi_{\text{AES, fct}} = (\text{in}, \text{out})$, $\nu(\text{in}) : \text{uint8}$, $\nu(\text{out}) : \text{uint24}$ and the functional specification $C^1_{\text{AES, fct}}$, we can imagine the function of the AES encoder according to Fig. 9.5, buffering three inputs of 8-bit and encoding them via $f_{\text{AES}}()$. As the pure functional contract $C^1_{\text{AES, fct}}$ does not determine the causality between inputs and outputs, a decision for the sequential or parallel execution of $f_{\text{AES}}()$ or w.r.t. the component's memory behavior is undetermined from this point of view. Providing the additional causality specification $f_{\text{order}}(i_{\text{in}}, i_{\text{out}}) := e^{\text{in}}_{3i_{\text{out}}-2} \sqsubset e^{\text{in}}_{3i_{\text{out}}-1} \sqsubset e^{\text{in}}_{3i_{\text{out}}} \sqsubset e^{i}_{\text{out}}(\text{out})$ resolves this, by appropriately mapping the indices to the common index i'.

$$C^1_{\text{AES,time}} \quad (\mathbf{x}^{in}_{\text{AES,time}} = (\mathbf{x}^{in}_{\text{AES,fct}}, f_{clk}); \nu(f_{clk}) : \text{uint8};) :$$
$$A^1_{\text{AES,time}} : f_{in}(t_i) = 1/8 f_{clk}; \quad \forall t_i = t(e^{in}_i) > 0;$$
$$G^1_{\text{AES,time}} : t(e^{out}_i)) \in \left[t(e^{in}_{3i}) + 30T_{clk}, t(e^{in}_{3i}) + 50T_{clk}\right];$$
$$\text{with} : f_x(t_i) := 1/T_x(t_i); \quad f_{clk} := 1/T_{clk};$$
$$T_x(t_i) := t(e^x_i) - t(e^x_{i-1}); \quad i = i(e^x_{t_i});$$

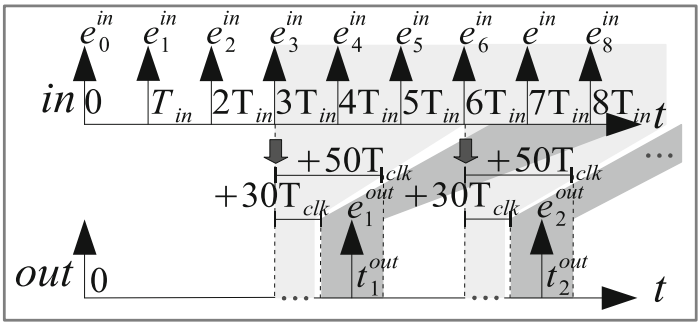


Fig. 9.6 AES timing specification via $C^1_{\text{AES, time}}$

The same ordering could be obtained from *timing contracts*, declaring the more detailed timing constraints w.r.t. the common *clock t*. As an example, depicted in Fig. 9.6, for the given timing specification $C^1_{\text{AES, time}}$, the AES component assumes inputs at a frequency of one eighth the clock frequency and promises to provide the corresponding output between thirty and fifty clock cycles later. Since both, the timing and the functional contract, demand the component to provide outputs for each of the input triples—consequently meaning to miss no inputs—the timing specification implicitly demands for a parallel decomposition of the f_{AES} encoding w.r.t. a buffering behavior for at least three to six inputs. According to this degree of freedom, the specification defines a set of time-to-time mappings, sketched by the grey shadings, and timing annotations in Fig. 9.6.

To finally constrain the implementation w.r.t. to its power consumption, a power contract $C^1_{\text{AES, pwr}}$ as given in Fig. 9.7 could demand the system not to exceed an average power consumption of e. g. 35 mW per encoding in power mode $\overrightarrow{\text{pm}}^{\text{in}}_1 = (1.0\,\text{V}, 100\,\text{MHz})$ resp. 60 mW per encoding in power mode $\overrightarrow{\text{pm}}^{\text{in}}_2 = (1.1\,\text{V}, 150\,\text{MHz})$ and 85 mW per encoding in power mode $\overrightarrow{\text{pm}}^{\text{in}}_3 = (1.1\,\text{V}, 200\,\text{MHz})$—all under the assumption of an arbitrary but time invariant *architecture* and *technology*.

9.8.2 AES Implementation and Characterization

Following steps 2–4 of Fig. 9.2, an implementation of the AES example can be designed from the given top-down specifications. Bottom-up, the resulting implementation can then be characterized w.r.t. timing and power and the results

$$C^1_{\text{AES,pwr}}\,(\mathbf{x}^{in}_{\text{AES,pwr}} = (\mathbf{x}^{in}_{\text{AES,time}}, V_{\text{DD}}, p_{\text{avg}});$$
$$\nu(V_{\text{DD}}) : \text{uint8}; \nu(p_{\text{avg}}) : \text{uint30};) :$$
$$A^1_{\text{AES,pwr}} : \nu(\overrightarrow{pm}, t) \in \{\{1000, 1100, 2000\}\text{mV} \times \{100, 150, 200\}\text{MHz}\};$$
$$B^1_{\text{AES,pwr}} : \nu(\overrightarrow{pm}, t) = (1000\text{mV}, 100\text{MHz}); \forall t(e_i^{\overrightarrow{pm}})) \in [t(e^{out}_{i-1}), t(e^{out}_i)];$$
$$G^1_{\text{AES,pwr}} : \nu(p_{\text{avg}}, t)(\mathbf{x}^{in}_{\text{pwr}}, t(e^{out}_{i-1}), t(e^{out}_i)) \leq 35\text{mW};$$
$$B^2_{\text{AES,pwr}} : \nu(\overrightarrow{pm}, t) = (1100\text{mV}, 150\text{MHz}); \forall t(e_i^{\overrightarrow{pm}}) \in [t(e^{out}_{i-1}), t(e^{out}_i)];$$
$$G^2_{\text{AES,pwr}} : \nu(p_{\text{avg}}, t)(\mathbf{x}^{in}_{\text{pwr}}, t(e^{out}_{i-1}), t(e^{out}_i)) \leq 60\text{mW};$$
$$B^3_{\text{AES,pwr}} : \nu(\overrightarrow{pm}, t) = (1200\text{mV}, 200\text{MHz}); \forall t(e_i^{\overrightarrow{pm}}) \in [t(e^{out}_{i-1}), t(e^{out}_i)];$$
$$G^3_{\text{AES,pwr}} : \nu(p_{\text{avg}}, t)(\mathbf{x}^{in}_{\text{pwr}}, t(e^{out}_{i-1}), t(e^{out}_i)) \leq 85\text{mW};$$

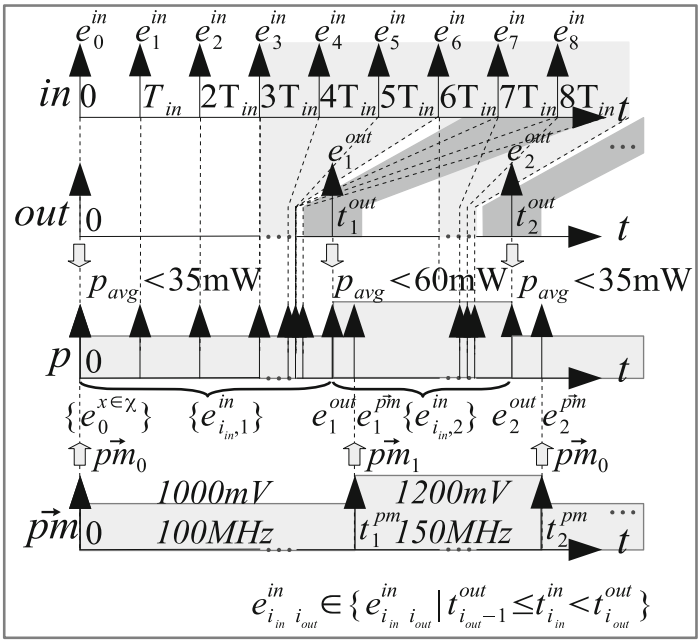


Fig. 9.7 AES power specification via $C^1_{\text{AES, pwr}}$

can be verified versus the contracts, to provide valid contracts for the bottom-up VI according to the steps 4–7 of Fig. 9.2.

For the example of the AES system, the initial functional high-level implementation derived from $C^1_{\text{AES, fct}}$ is drafted in Fig. 9.8, using $f_{\text{AES}}()$ to abbreviate the encoding function and $[\langle \ldots \rangle]$ for the guards resp. $/\langle \ldots \rangle$ for the updates of the edges.

For the computation of $f_{\text{AES}}()$ we assume the availability of two possible Encoder solutions ENC_1 and ENC_2, providing the extra-functional properties according to Table 9.1. Hence, we have no component, computing f_{AES} in less then $23T_{\text{clk}}$ and the timing specification $C^1_{\text{AES, time}}$ would not be satisfiable without a parallel

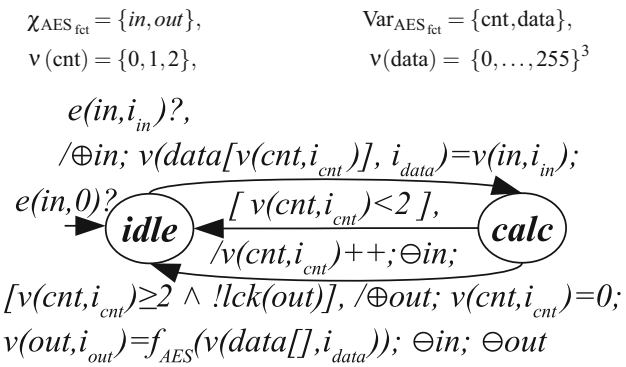


Fig. 9.8 Initial implementation of the functional specification $C^1_{\text{AES, fct}}$

Table 9.1 Timing and power characterization of the AES Encoder and Buffer components' functional states q.

HRC	q	$t_q/[\#T_{\text{clk}}]$	$\bar{C}_{\text{sw}}(q)/[pF]$
BUF	Idle	–	80
	Buffer	4	123
	Send	7	109
ENC_1	Idle	–	10
	Encode	506	106
ENC_2	Idle	–	10
	Encode	23	234

decomposition between f_{AES} and additional buffering. As a result, the initial implementation in Fig. 9.8 is functionally refined as a composition of two HRCs: *Buffer* (*BUF*) and *Encoder* (*ENC*), according to Fig. 9.9, communicating via the delegation connector net(BUF.out, ENC.in), which mutually identifies BUF.out with ENC.in and vice versa, using the *port substitutions* ρ_{BUF} and ρ_{ENC}, resulting in the equality of those traces' values, events, and semaphores.

For a dynamic power modeling, depending on the timed functional state as well as the timed voltage and frequency modes, all edges into a power state q_{ps} are extended with an update function $set\_p(\bar{C}_{\text{sw}}(q_{\text{ps}}))$, which calculates the power states' current power consumption $p(\mathbf{x}^{\text{in}}_{\text{pwr}}, t) = v^2(V_{\text{DD}}, i) \cdot v(f_{\text{clk}}, i) \cdot \bar{C}_{\text{sw}}(q_{\text{ps}})$; based on their characteristic switched capacitance and the current values of voltage and frequency, writing the result to the power ports p_{BUF} and p_{ENC} and triggering the appropriate trace events $e(p_{\text{BUF}}, i)$ or $e(p_{\text{ENC}}, i)$.

Finally, the resulting UPPAAL HRC, derived from interweaving the models of function, timing, and power consumption in a combined multi aspect model of the exemplary AES system, re-using the Encoder implementation ENC_2, is given in Fig. 9.9. Showing the complete extra-functional behavior at the bottom, the interweaving of only the functional and the timing behavior is completely equivalent, omitting only the update functions $set_p(\bar{C}_{\text{sw}}(q_{\text{ps}}))$.

$$\chi_{\mathrm{AES}_{hrc}} = \{in, out, f_{clk}, V_{\mathrm{DD}}, p_{\mathrm{AES}}\},$$
$$\mathrm{Var}_{\mathrm{AES}_{hrc}} = \{\mathrm{cnt}, \mathrm{data}, C_{sw,\mathrm{BUF}}, C_{sw,\mathrm{ENC}}\},$$
$$\nu(\mathrm{cnt}) = \{0,1,2\}, \nu(\mathrm{data}) = \{0,\ldots,255\}^3,$$
$$\nu(C_{sw,\mathrm{BUF}}) = \nu(C_{sw,\mathrm{ENC}}) = \mathbb{N}_0^+, \mathrm{Clks}_{\mathrm{AES}_{hrc}} = \{t_{\mathrm{BUF}}, t_{\mathrm{ENC}}\}$$

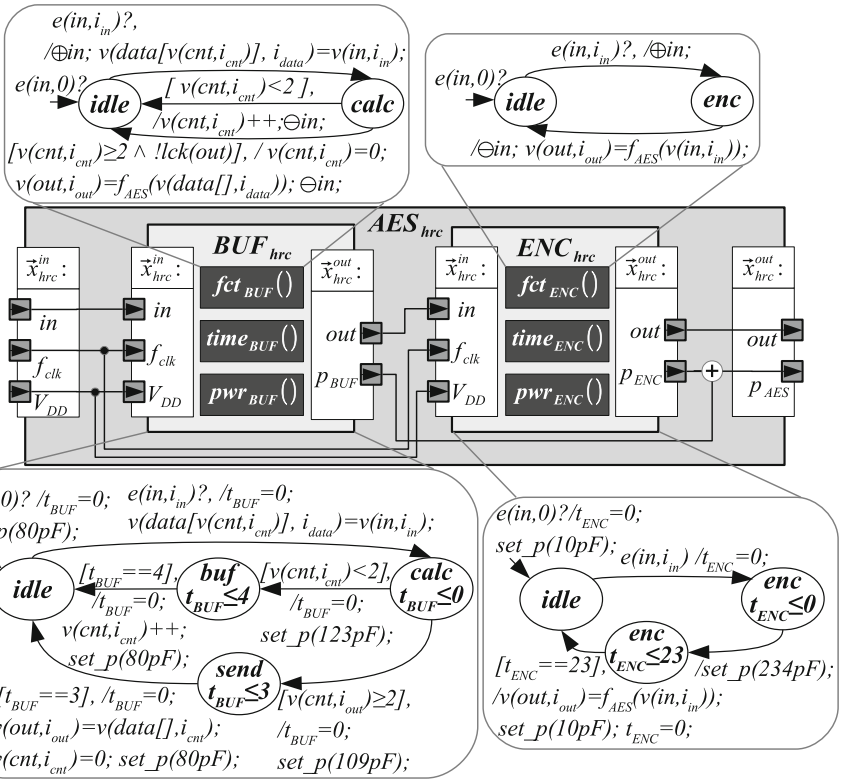
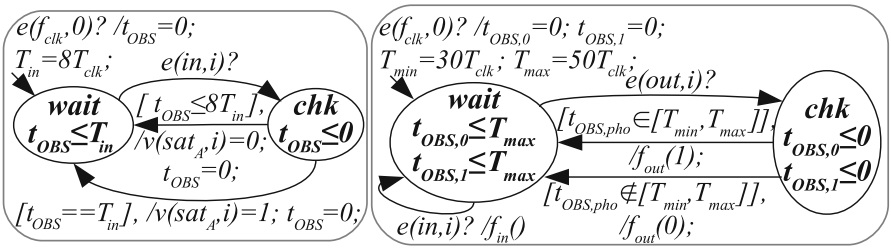


Fig. 9.9 Resulting multi aspect refinement of the AES, showing the structural bottom-up composition of the HRCs BUF and ENC in the middle, their basic functional implementations at the top and their combined multi aspect power behavior—interweaving the functional implementations with the components' timing and power characteristics according to Sect. 9.6 at the bottom

9.8.3 AES Verification

To validate if the multi aspect implementation holds the contracts, the contracts are expressed by a combination of (UPPAAL-RSL) expressions and timed observer automata. Focussing the extra-functional aspects of power contracts we omit the functional verification to concentrate on the verification of the timing and power behavior. For checking the timing contract $C^1_{\mathrm{AES,\,time}}$, two observer automata are implemented according to Fig. 9.10: $\mathrm{OBS}^1_{A,\mathrm{time}}$ (left) to verify the assumption

$$\chi_{\mathrm{OBS}} = \{in, out, f_{clk}, sat_A, sat_G\}, \mathrm{Clks}_{\mathrm{OBS}} = \{t_{\mathrm{OBS}}, t_{\mathrm{OBS},0}, t_{\mathrm{OBS},1}\},$$
$$\mathrm{Var}_{\mathrm{OBS}} = \{\mathrm{cnt}_0, \mathrm{cnt}_1, phi, pho, T_{in}, T_{\min}, T_{\max}\},$$
$$\nu(\mathrm{cnt}_0) = \nu(\mathrm{cnt}_1) = \{0,1,2\}, \nu(T_{in}) = \nu(T_{\min}) = \nu(T_{\max}) = \mathbb{N}_0^+,$$
$$\nu(phi) = \nu(pho) = \nu(sat_A) = \nu(sat_G) = \mathbb{B}$$



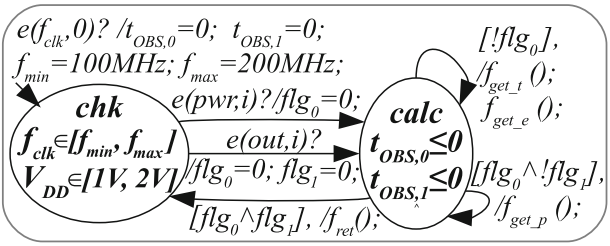
$$f_{in}()\,\{\,\mathrm{if}(\mathrm{cnt}_{phi} < 2)\,\mathrm{cnt}_{phi} = \mathrm{cnt}_{phi} + 1;$$
$$\mathrm{else}\,\{\mathrm{cnt}_{phi} = 0; t_{\mathrm{OBS},phi} = 0; phi = !phi;\,\}\}$$
$$f_{out}(\mathrm{bool}\,t)\,\{\,sat_G = t; t_{\mathrm{OBS},pho} = 0; pho = !pho;\,\}$$

Fig. 9.10 Observer TAs $\mathrm{OBS}_{A^1_{\mathrm{AES,time}}}$ (*left*) and $\mathrm{OBS}_{G^1_{\mathrm{AES,time}}}$ (*right*) to verify the assumption $A^1_{\mathrm{AES,time}}$ and the guarantee $G^1_{\mathrm{AES,time}}$ of contract $C^1_{\mathrm{AES,time}}$

$A^1_{\mathrm{AES,time}}$, i.e., the frequency of the inputs; and $\mathrm{OBS}^1_{G,\mathrm{time}}$ (right) to verify the associated guarantee $G^1_{\mathrm{AES,time}}$, i.e., the I/O delay between every third input and its corresponding output. For simplification we onwards deviate from the original UPPAAL syntax, using functions and pseudo code to sketch the effect of more complex parts of the automata. First, in $\mathrm{OBS}^1_{G,\mathrm{time}}$ we use $f_{\mathrm{in}}()$ to detect and alternately control the start of the TA's two different clocks $t_{\mathrm{OBS},0}$ and $t_{\mathrm{OBS},1}$ using the Boolean flag *phi*. Then, complementing $f_{\mathrm{in}}()$ w.r.t. to the allowed interleaving of inputs and outputs, $f_{\mathrm{out}}()$ is used to set the verification result sat_G and to control the end of the current clock's verification cycle using the Boolean flag *phi*.

Finally, when the functional and the timing contracts are valid, the power contract $C^1_{\mathrm{AES,pwr}}$ can be verified by the observer given in Fig. 9.11. Here, to abbreviate the use of multiple edges, we summarize the events $e(p_{\mathrm{BUF}}, i)$ and $e(p_{\mathrm{ENC}}, i)$ by the abstract event $e(\mathrm{pwr}, i)$, which controls the first clock $t_{\mathrm{OBS},0}$, responsible for measuring the duration between any changes of the power consumption, and appropriately starts the second clock $t_{\mathrm{OBS},1}$, which measures the duration of the contract's averaging cycle between the occurrence of the first input and the corresponding output event. Then, to enable the clock value for arithmetic calculations, the abstraction $f_{\mathrm{get\_}t}()$ represents another part of the automaton, which converts $t_{\mathrm{OBS},0}$ to an equivalent integer variable t_0. On that base, $f_{\mathrm{get\_}e}()$ computes the system's energy consumption during the preceding time interval t_0, using the previous values $p_{\mathrm{BUF},p}$ and $p_{\mathrm{ENC},p}$ of the components' power traces. When the averaging cycle of $C^1_{\mathrm{AES,pwr}}$ ends with the output event $e(\mathrm{out}, i)$, the second clock t_1 is stopped and similarly converted to integer inside the abstract function $f_{\mathrm{get\_}p}()$, where it is used

$$\chi_{\text{OBS}} = \{in, out, f_{clk}, V_{\text{DD}}, p_{\text{BUF}}, p_{\text{ENC}}, sat\}, \text{Clks}_{\text{OBS}} = \{t_{\text{OBS},0}, t_{\text{OBS},1}\},$$
$$\text{Var}_{\text{OBS}} = \{t_0, t_1, \text{flg}_0, \text{flg}_1, f_{\min}, f_{\max}, v_{\min}, v_{\max}, e, p_{\text{BUF},p}, p_{\text{ENC},p}\},$$
$$\nu(\text{flg}_0) = \nu(\text{flg}_1) = \nu(sat) = \mathbb{B},$$
$$\nu(t_0) = \nu(t_1) = \nu(e) = \nu(p_{\text{BUF},p}) = \nu(p_{\text{ENC},p}) = \mathbb{N}_0^+$$



$$f_{ret}() \{ sat = (p_{\text{OBS}} \leq p_{\text{avg}}); p_{\text{BUF},p} = p_{\text{BUF}}; p_{\text{ENC},p} = p_{\text{ENC}}; \}$$
$$f_{get\_t}() \{ t_0 = t_{\text{OBS},0}; t_{\text{OBS},0} = 0; \}$$
$$f_{get\_e}() \{ e = e + t_0 \cdot (p_{\text{BUF},p} + p_{\text{ENC},p}); \text{flg}_0 = 1; \}$$
$$f_{get\_p}() \{ t_1 = t_{\text{OBS},1}; p_{\text{OBS}} = e/t_1; e = 0; t_{\text{OBS},1} = 0; \text{flg}_1 = 1; \}$$

Fig. 9.11 Observer TA to verify the assumption $A^1_{\text{AES, pwr}}$ and guarantee $G^1_{\text{AES, pwr}}$ of the contract $C^1_{\text{AES, pwr}}$

for averaging the accumulated energy e over the interval t_1 to obtain the average power consumption $p_{\text{OBS}} = p_{\text{avg, AES}}(\mathbf{x}^{\text{in}}_{\text{pwr}}, t_1)$. Finally, if the resulting average holds the average power specification of the power contract, *sat* evaluates to *true*.

Verifying the AES example for $v(\overrightarrow{\text{pm}}) = (1000\,\text{mV}, 100\,\text{MHz})$ or $v(\overrightarrow{\text{pm}}) = (1100\,\text{mV}, 150\,\text{MHz})$ and with the appropriate input frequency of $v(f_{\text{in}}) = 1/8 f_{\text{clk}}$ the functional, timing, and power contracts are satisfied, being a valid bottom-up formalization of the system. In contrast, driving the same system at $v(\overrightarrow{\text{pm}}) = (1200\,\text{mV}, 200\,\text{MHz})$ the average power consumption exceeds 85 mW to a maximum of 94 mW. According to the exhaustiveness of model checking the AES for all configurations of its environment—that is especially for all specified power modes—the invalid extra-functional re-use can be identified, enabling for an improvement of the components' implementations resp. for a correction of the formal bottom-up power characterization, providing a weaker guarantee.

9.9 Conclusion

Since energy consumption has become one of the most limiting factors of today's embedded and integrated systems we investigate contract-based design to build a consistent methodology for power-aware system design. Integrating the functional, timing, and power aspects of a bottom-up component characterization, we derive their corresponding heterogeneous component models in UPPAAL, enabling them

for an exhaustive, formal verification between their implementation and their corresponding, extra-functional specification, based on contracts. To transfer our idea of power contracts into analyzable UPPAAL syntax, we use a combination of UPPAAL-RSL expressions and additional timed observer automata. Applying power contracts to the example of a composed AES system, we successfully analyzed an initial proof of concept, allowing for the leaf-node verification of power contracts in UPPAAL.

The next steps to substantiate and improve the presented methodology will now contain a further generalization and formalization of integrating the components' aspect specific implementations as well as of generating analyzable UPPAAL observer from a sufficiently generic set of power contracts. Furthermore, we proceed with investigating power contracts w.r.t. the subsequent virtual integration process.

Acknowledgements This work has been supported by the EnerSave Project, funded by the German Federal Ministry of Research and Education (BMBF) under Grant Agreement 16BE1102 and the FP7 project CONTREX, funded by the European Commission under Grant Agreement 611146.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**(2) (1994). doi: 10.1016/0304-3975(94)90010-8
2. Behrmann, G., David, A., Larsen, K.G.: A tutorial on UPPAAL. In: Formal Methods for the Design of Real-Time Systems. Springer, Berlin (2004)
3. Ben Abdallah, F., Apvrille, L.: Fast evaluation of power consumption of embedded systems using DIPLODOCUS. In: 39th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA'13) (2013). doi:10.1109/SEAA.2013.8
4. Bengtsson, J., Yi, W.: Timed automata: semantics, algorithms and tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) Lectures on Concurrency and Petri Nets, Lecture Notes in Computer Science, vol. 3098. Springer, Berlin/Heidelberg (2004)
5. Benveniste, A., Caillaud, B., Nickovic, D., Passerone, R., Raclet, J.B., Reinkemeier, P., Sangiovanni-Vincentelli, A., Damm, W., Henzinger, T., Larsen, K.: Contracts for systems design. Technical Report RR-8147, Research Centre Rennes—Bretagne Atlantique, Rennes Cedex (2012)
6. Boudjadar, J., David, A., Kim, J.H., Larsen, K.G., Mikucionis, M., Nyman, U., Skou, A.: Schedulability and energy efficiency for multi-core hierarchical scheduling systems. In: Proceedings of the Conference on Embedded Real Time Systems and Software (ERTSS'14). Toulouse (2014)
7. Damm, W., Dierks, H., Oehlerking, J., Pnueli, A.: Towards component based design of hybrid systems: Safety and stability. In: Manna, Z., Peled, D.A. (eds.) Time for Verification Essays in Memory of Amir Pnueli. Lecture Notes in Computer Science, vol. 6200. Springer, Berlin/Heidelberg (2010)
8. Damm, W., Hungar, H., Josko, B., Peikenkamp, T., Stierand, I.: Using contract–based component specifications for virtual integration testing and architecture design. In: Lukasiewycz, M., Chakraborty, S., Milbredt, P. (eds.) Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011. Grenoble, France (2011). doi:10.1109/DATE.2011.5763167

9. David, A., Larsen, K.G., Legay, A., Nyman, U., Wasowski, A.: Timed I/O automata: a complete specification theory for real-time systems. In: Proceedings of the 13th ACM international conference on Hybrid systems: computation and control, HSCC '10. New York (2010). doi: 10.1145/1755952.1755967
10. Gomez, C., DeAntoni, J., Mallet, F.: Power consumption analysis using multi-view modeling. In: Proceedings of the 23rd International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS). IEEE, Karlsruhe (2013)
11. Holzmann, G.J.: Design And Validation Of Computer Protocols. Prentice Hall, New Jersey (2007)
12. Hungar, H.: Components and contracts: a semantical foundation for compositional refinement. In: Tagungsband MBEES: Modellbasierte Entwicklung eingebetteter Systeme 2012 (2012)
13. IEEE Computer Society, Design Automation Committee, IEEE Standards Association, Corporate Advisory Group, Institute of Electrical and Electronics Engineers, IEEE-SA Standards Board: IEEE Standard for Design and Verification of Low-Power Integrated Circuits. IEEE Standards Association, New Jersey (2013)
14. Josko, B., Ma, Q., Metzner, A.: Designing embedded systems using heterogeneous rich components. In: Proceedings of the INCOSE International Symposium (2008)
15. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. In: Proceedings of the 2nd Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science, vol. 1055. Springer, Berlin (1995)
16. Lee, E.A., Sangiovanni-vincentelli, A.: The tagged signal model - a preliminary version of a denotational framework for comparing models of computation. Technical Report UCB/ERL M96/33, University of California, Berkeley, CA (1996)
17. Lee, E., Sangiovanni-Vincentelli, A.: A framework for comparing models of computation. IEEE Trans. Comput.-Aided Design Integr. Circuits Syst. **17**(12) (1998). doi:10.1109/43.736561
18. Liu, X.: Semantic foundation of the tagged signal model. Ph.D. thesis, EECS Department, University of California, Berkeley, (2005)
19. Lorenz, D., Grüttner, K., Bombieri, N., Guarnieri, V., Bocchio, S.: From RTL IP to functional system-level models with extra-functional properties. In: Jerraya, A., Carloni, L.P., Chang, N., Fummi, F. (eds.) Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS '12. ACM, New York, NY (2012). doi:10.1145/2380445.2380529
20. Lorenz, D., Hartmann, P.A., Grüttner, K., Nebel, W.: Non-invasive power simulation at system-level with SystemC. In: Ayala, J.L., Shang, D., Yakovlev, A. (eds.) Proceedings of the International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS) 2012. Lecture Notes in Computer Science, vol. 7606. Springer, Newcastle upon Tyne (2012)
21. Nitsche, G., Grüttner, K., Nebel, W.: Power contracts: a formal way towards power–closure?! In: Proceedings of the 23rd International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS'13). IEEE, Karlsruhe (2013)
22. Sangiovanni-Vincentelli, A., Damm, W., Passerone, R.: Taming Dr. Frankenstein: contract-based design for cyber-physical systems. Eur. J. Control **18**(3) (2012). doi:10.3166/ejc.18.217-238
23. Silicon Integration Initiative, I.S.: Si2 Common Power Format Specification™, version 2.0 edn. Silicon Integration Initiative, Inc. (Si2™) (2011)
24. UPPAAL: http://www.uppaal.org/
25. Urdahl, J., Stoffel, D., Kunz, W.: Path predicate abstraction for sound system-level models of RT-level circuit designs. IEEE Trans. Comput.-Aided Design Integr. Circuits Syst. **33**(2) (2014). doi:10.1109/TCAD.2013.2285276

Chapter 10
SystemC AMS Power Electronic Modelling with Ideal Instantaneous Switches

Leandro Gil and Martin Radetzki

Abstract Ideal instantaneous switches are a useful behaviour abstraction technique for modelling semiconductor components in power system development. This behaviour abstraction allows fast and robust simulations of sophisticated power systems. In this paper, we present a SystemC AMS extension that supports ideal switches modelling and simulation. Using this extension, large externally and internally controlled electrical linear networks can be integrated into system level models for design and verification purposes. To validate our implementation, we modelled and simulated a complex high voltage power converter for medical applications. The results demonstrate the robustness and accuracy of our SystemC AMS extension.

10.1 Introduction

System level design and verification of analogue and mixed-signal hardware and software requires a hierarchical approach that uses different levels of abstraction. SystemC is a system level design language (SLDL) what focuses on system architecture design for large systems. It provides an open environment for consistent and efficient modelling and simulation of complex heterogeneous systems.

Based on a set of C++ classes and methods, the structure and the behaviour of hardware and software systems can be described from abstract specifications to register transfer level (RTL).

Analogue circuit modelling at higher levels of abstraction is gaining importance in facilitating high-performance system-level simulations. In order to support complex System-on-Chip (SoC) design SystemC was extended for analogue and mixed signal aspects. The current SystemC AMS standard includes models of computation (MoCs) for continuous and discrete time data flow modelling as well as conservative behaviour descriptions for electrical network modelling. These

L. Gil (✉) • M. Radetzki
Embedded Systems Department, University of Stuttgart, Pfaffenwaldring 5b,
70569 Stuttgart, Germany
e-mail: leandro.gil@informatik.uni-stuttgart.de; martin.radetzki@informatik.uni-stuttgart.de

© Springer International Publishing Switzerland 2016

F. Oppenheimer, J.L. Medina Pasaje (eds.), *Languages, Design Methods, and Tools for Electronic System Design*, Lecture Notes in Electrical Engineering 361,
DOI 10.1007/978-3-319-24457-0_10

abstraction methodologies provide enough facilities to support system description and simulation for a wide range of applications, especially for communication systems.

For applications requiring large signal behaviour and switching mode operation, such as driver stages with pulse width modulation, the existing MoCs do not achieve the necessary accuracy [18] or simulation performance [10, 17]. To overcome these limitations, we propose a SystemC AMS extension for power electronic modelling that relies on ideal instantaneous switches. This extension provides primitives for modelling internally and externally controlled switched networks and enables fast, robust and accurate simulations.

The presented approach implements new primitives for semiconductor modelling and reduces the size of system equations by exploiting the properties of ideal switched electrical networks. Additionally, a set of libraries are provided for validating the power circuit functionality in Simulink [8]. Electrical circuits described using SystemC AMS syntax can then be embedded in user code and thus integrated smoothly into Simulink models. The presented methodology does not intend to replace specialized analogue simulators such as Simulink/PLECS or SPICE. These programs are advantageous because they provide a set of specialized libraries and development tools to support several applications and they are normally faster or more accurate when systems with one large analogue part are simulated.

In Sect. 10.2, we outline previously implemented SystemC extensions supporting electrical network modelling and investigate similar modelling approaches for continuous time simulators. Section 10.3 introduces the modelling of switched networks. Using a simple example, some limitations of the current SystemC AMS standard for power electronic modelling are then illustrated in Sect. 10.4. The proposed model of computation and its computational implementation are described in the Sects. 10.5 and 10.6. Finally, we demonstrate the practical applicability of the proposed extension using a sophisticated industrial example in Sect. 10.7.

10.2 Related Work

10.2.1 SystemC Extensions Supporting Electrical Networks

Al-Junaid and Kazmierski presented a SystemC extension named SEAMS using a general-purpose analogue solver [1–3]. In order to provide modelling capabilities for general, mixed-mode systems with digital and non-linear analogue behaviour, a variety of abstraction levels, from system level to circuit level were proposed in this work. The described language constructs support analogue system variables, analogue components and user defined ordinary differential and algebraic equations. C++ classes for electrical nodes and primitive analogue components such as resistor, capacitor, inductor, diode and various types of sources have been implemented. An electrical circuit can be constructed by declaring system variables of type node and analogue components. At the matrix build time, the build functions of all components in the *net list* are invoked.

The analogue equation set formulation relies on the modified nodal analysis (MNA). The analogue kernel invokes the build function once before simulation start. For synchronization between SystemC ports and their corresponding values in the analogue solver, interfacing components are provided. The SystemC kernel was extended to invoke and synchronize the analogue solver in each simulation cycle by applying a lock-step method. A boost power converter was chosen as example of non-trivial analogue and digital interaction. Any numerical difficulty was reported. In a second implementation, named SystemC-A, the handling of extremely small and zero time steps was incorporated to deal with simulation issues.

Vachoux, Grimm and Einwich presented the core elements of SystemC AMS in [19] and [11]. To fill the gap in heterogeneous SoC modelling and simulation, the SystemC extension for analogue and mixed-signal systems was focused on signal processing, RF/wireless and automotive applications. It includes features for modelling linear dynamic continuous time systems and linear networks. In order to support true object oriented model refinement, from abstract specifications to detailed implementations, generalized signals and channels were defined. Using a static dataflow scheduler (with fixed time steps in the first release) synchronization between continuous-time and discrete event model parts was achieved. In a first approach continuous-time descriptions were embedded into discrete-event modules as a cluster of dataflow components.

SystemC AMS is structured using a layered approach. New continuous time models of computation can be added utilizing the descriptive methods provided by the user view layer. Different solver implementations are possible at the solver layer. Finally, the synchronization layer implements a generic mechanism to interface continuous-time solvers and discrete event parts.

To enable the modelling of analogue, non-linear parts of cyber-physical systems at higher levels of abstraction, Uhle and Einwich proposed in [18] a new model of computation for SystemC AMS with similar features like in VHDL-AMS or VerilogAMS. It allows the modelling and simulation of nonlinear networks. The proposed extension integrates smoothly into the existing SystemC AMS language architecture.

The previous three approaches implement models of computation for a wide range of applications. Although electrical network modelling is supported by all theses extensions, they don't provide the necessary accuracy or simulation performance for the modelling of power electronic systems at high abstraction levels. SystemC AMS allows rapid electrical network simulations, but it does not offer primitives for the modelling of internally controlled switches. The applied numerical integration method limits the type of circuit that can be simulated. The proposed extension for nonlinear continuous behaviour leads to sophisticated models and slow simulations. It restricts the practical application to small power circuits.

In [13], Grimm et al. presented a novel approach to enable fast simulations of analogue power drivers. The C++ behavioural models can be easily integrated in SystemC. The underlying method utilizes pre-solved parameterized differential equations. The dominant cycle of the network is determined using Dijkstra's

algorithm. Because, in this approach, analogue models cannot cause discrete events, only a single topological change is permitted at each switching instant. It is not possible to simulate internally controlled switches such as diodes.

10.2.2 Electrical Networks with Ideal Switches

Bedrosian and Vlach presented in [6, 7] a model of computation for time domain analysis of networks with internally controlled ideal switches. In this work the network equations for each topology are generated with a two-graph MNA technique. In order to determine the correct topology after switching, impulsive voltages and currents are considered at the switching instants, in conjunction with the initial conditions. An accurate handling of voltage and current impulses at the switching instants is carried out by splitting the circuit response into a non-impulsive and an impulsive component. The impulsive response part is calculated using a computational method based on inverse Laplace transformation. To find a valid topology after switching several topological changes are allowed. This general analysis method is suitable for any internally controlled switched network (see Sect. 10.3). The computational approach was implemented in a circuit simulator named SWANN.

Massarini, Reggiani and Kazimierczuk proposed a method for large-signal time-domain analysis of switched networks based on a state variable approach in [12–14]. In their work, the network equations are represented with a reduced tableau matrix. An efficient algorithm for the systematic formulation of state equations and output equations for linear active networks was developed. Every switching element is also modelled as an ideal switch. The evolution of the network is represented by a sequence of linear circuit topologies. For each topology, the state equations are systematically obtained through a simple interchange of columns. In order to find the correct topology after switching, a logical representation of impulsive quantities is introduced in the analysis. Using the presented state space description a method to predict possibly impulsive transitions was presented. The impulse analysis is performed only for a limited number of transitions. Switched networks consisting of linear elements and both externally and internally controlled switches were simulated.

Based on the previous approaches Allmeling and Hammer developed a toolbox, PLECS, for simulation of power electronic circuits under Simulink (See [4, 5, 15, 16]). It exploits the features offered in the Simulink environment, allowing the simulation of large systems containing both electrical circuits and sophisticated controllers. A state-space circuit formulation is also applied in this work. The independent mesh and node equations are, however, obtained using MNA. The automatically generated equation system, describing the circuit, is then reduced by elimination of dependent variables. In order to derive the state matrices for a specific topology, the system variables are ordered into: state derivatives, output variables, undetermined switch variables, state variables and input variables. Using Gauss

Jordan elimination with partial pivoting, the generic equation system is transformed into an upper triangular matrix. The state-space equations are then embedded in Simulink by means of user code in C. By analysing the system outputs and the control inputs a switch manager determines the circuit topology. In addition to the non-impulsive part of the system output, an output impulse-multiplier, which is implicitly associated with a voltage or current impulse, is determined after switching. In order to hit the exact switching point in time, Simulink's zero crossing detector is utilized.

Unfortunately, PLECS does not provide a toolbox for discrete event simulators such as Verilog or SystemC. In order to carry out power circuit simulations in SystemC, we exploit the generic architecture of SystemC AMS to develop a model of computation supporting internally and externally controlled ideal switches. A special solver enabling instantaneous switching and accurate results for discrete time simulation is presented in this work.

10.3 Power Electronic Modelling

10.3.1 Abstraction of Power Electronic Circuits

Depending on design and verification goals, different abstraction levels are commonly used to model power electronic circuits:

1. Transfer function:
   For controller design a mathematical description of the system in form of a transfer function is commonly used. The electric circuit is considered as linear causal system. It is only valid for small signal behaviour and no switching response can be analysed (no harmonics).
2. Electrical network with ideal components:
   For circuit design and controller verification tasks a time domain mathematical description of the system is required. The electric circuit is represented using linear components and switches. It is valid for large signal behaviour and applied to evaluate the overall system performance. Voltage and current waveforms of different system parts are analysed.
3. Electrical network with detailed components:
   At the last design stages, a detailed mathematical model of circuit components, including manufacturer specific characteristics, is required for the choice of components. Parasitic effects, switching transitions and component stress are considered during the analysis. Usually, non-linear component behaviour needs to be considered. SystemC AMS provides MoCs to describe analogue systems as transfer function or electrical network. Circuit specific characteristics cannot be adequately analysed. A specialized circuit simulator such as SPICE or SABER is required for such tasks.

10.3.2 Classes of Switched Networks

Taking into consideration the network components and topology, according to [7] switched networks can be classified into:

1. Externally controlled switched networks:
   In this type of electrical networks the state of the switches does not depend on the network response. All switches are controlled by external signals. It leads to "forced commutations".
2. Internally controlled switched networks:
   In this type of electrical networks one or more switches are controlled by network voltages and/or currents. The switching time between different topologies depends on the state of network components. It leads to "natural (and forced) commutations".

SystemC AMS does not include primitives for internally controlled switches. They can be modelled by implementing their control logic as separated module. Alternatively, user primitives can be developed from basic SystemC AMS classes. Additionally, the handling of multiple simultaneous switching requires an appropriated extension, as will be explained in the next chapter.

10.4 Limitations of Modelling and Simulating Power Electronics in SystemC AMS

Power electronic circuits commonly consist of linear components and one or more semiconductor switches. These elements are already included in the ELN primitives. The current computational approach of this MoC provides very good results for the applications considered in the SystemC AMS specification, namely signal processing, RF/wireless and automotive [19].

As mentioned in [9], there are some limitations regarding the simulation of electrical linear networks. Some networks can be described using ELN primitives but the resulting differential equation system cannot be solved.

The Buck converter circuit shown in Fig. 10.1 wasutilized in [4, 5] to explain ideal switching modelling. It is useful to illustrate current SystemC AMS limitations for modelling and simulation of power electronic models. Figure 10.2 shows the ELN code of the Buck converter circuit.

Figure 10.3 shows the ELN code of the diode. It is modelled using a resistive switch. Additional primitives for voltage and current monitoring (V_m and A_m) are required to represent the diode characteristics. The diode control logic is implemented using a simple discrete event module (Fig. 10.4).

When the presented model is simulated, an error message is displayed in the system console (see Fig. 10.5). The SystemC AMS solver cannot initialize the underlying equation system.

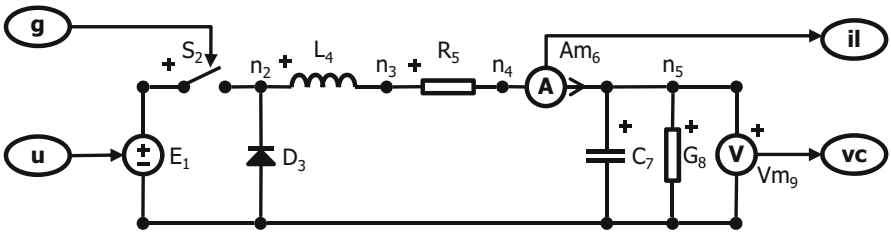


Fig. 10.1 Buck converter schematic

```
namespace sca_eln_mdl
{
    using namespace sca_eln;
    SC_MODULE(Circuit)
    {
        sc_core::sc_in<bool> g;
        sc_core::sc_in<double> u;
        sc_core::sc_out<double> i_l;
        sc_core::sc_out<double> v_c;

        sca_r R;    sca_l L;    sca_c C;    sca_r G;
        sca_de::sca_vsource E;  sca_de::sca_isink Am; sca_de::sca_vsink Vm;
        sca_de::sca_switch S; sca_de::sca_diode D;

        Circuit(sc_core::sc_module_name name):
            g("g"), u("u"), i_l("i_l"),
            R("R", 0.05), L("L",0.1), C("C",0.1), G("G",0.1),
            E("E",1), Am("Am"),Vm("Vm"),
            S("S"), D("D"),
            gnd("gnd"),n1("n1"),n2("n2"),n3("n3"),n4("n4"),n5("n5")
        {
            sc_core::sc_time time_step = sc_core::sc_time(1,sc_core::SC_US);
            E.p(n1); E.n(gnd); E.inp(u);        E.set_timestep(time_step);
            S.p(n1); S.n(n2); S.ctrl(g);        S.set_timestep(time_step);
            D.p(gnd); D.n(n2);                  D.set_timestep(time_step);
            L.p(n2); L.n(n3);R.p(n3); R.n(n4);
            Am.p(n4); Am.n(n5); Am.outp(i_l);   Am.set_timestep(time_step);
            C.p(n5); C.n(gnd); G.p(n5); G.n(gnd);
            Vm.p(n5); Vm.n(gnd); Vm.outp(v_c);  Vm.set_timestep(time_step);
        }
        private:
        sca_node_ref gnd;
        sca_node     n1, n2, n3, n4, n5;
    };
}
```

Fig. 10.2 SystemC AMS ELN Buck converter implementation

In order to keep the size of resulting system matrix constant, switches are described in SystemC AMS by a variable resistance. Very small and very large resistance values are respectively utilized to represent the ON and OFF switch states. This allows the exploitation of the fast and generic linear solver developed for continuous time modelling. No solver extension is required to compute the linear network solution after a topology change.

The circuit response usually does not change considerably if switches are modelled in this way; however, it often leads to non-solvable equation systems.

```
namespace sca_eln
{
    namespace sca_de
    {
        using namespace sca_eln::sca_de;
        SC_MODULE(sca_diode)
        {
            sca_eln::sca_terminal p;
            sca_eln::sca_terminal n;

            private:
            bool state; double r_on; double r_off;
            sca_rswitch D;  sca_isink Am; sca_vsink Vm;
            sca_diode_control ctrl;

            public:
            sca_diode(sc_core::sc_module_name,double _r_on = 0.0,
            double _r_off = sca_util::SCA_INFINITY):
            p("p"),n("n"),r_on(_r_on),r_off(_r_off), D("D",_r_on,_r_off),
            Am("Am"),Vm("Vm"), ctrl("ctrl"), n1("n1")
            {
                Am.p(p); Am.n(n1); Am.outp(i_d);        // Current measure
                D.p(n1); D.n(n); D.ctrl(s_d);           // Resistive switch
                Vm.p(p); Vm.n(n); Vm.outp(v_d);         // Voltage measure
                ctrl.v(v_d); ctrl.i(i_d); ctrl.s(s_d);  // Diode control module
            }

            void set_timestep(int time_step_value, sc_core::sc_time_unit time_step_unit)
            {
                set_timestep(sc_core::sc_time(time_step_value,time_step_unit));
            }

            void set_timestep(sc_core::sc_time _time_step)
            {
                time_step = _time_step;
                Am.set_timestep(time_step);
                D.set_timestep(time_step);
                Vm.set_timestep(time_step);
                ctrl.set_timestep(time_step);
            }

            void set_ron (double r_value){D.ron  = r_value;}
            void set_roff(double r_value){D.roff = r_value;}

            private:
            sc_signal<bool>   s_d;
            sc_signal<double> v_d, i_d;
            sca_eln::sca_node n1;
            sc_core::sc_time time_step;
        };
    }   // namespace sca_de
}       // namespace sca_eln
```

Fig. 10.3 SystemC diode implementation

By setting the diode switch off state resistance to a smaller value (1e9), the solvability problem can be eliminated.

The power circuit controller is also implemented using a discrete event module as shown in Fig. 10.6. The inductor current is maintained inside a defined range by open and closing the switch S. When the maximal inductor current is achieved, the controller opens the switch and a new solvability error occurs during simulation, as shown in Fig. 10.7. At the switching instant, the inductor current is interrupted causing a voltage impulse. This turns the diode ON. Due to the very small diode ON resistance, the resulting equation system cannot be solved. Changing diode ON resistance to an appropriated value (1e-6), the system equations can be numerically solved.

```
namespace sca_eln
{
    namespace sca_de
    {
        SC_MODULE(sca_diode_control)
        {
            public:
            sc_core::sc_in<double> v;
            sc_core::sc_in<double> i;
            sc_core::sc_out<bool> s;

            void event_control(void)
            {
                double voltage = v.read();
                double current = i.read();

                if((state==true)&&(current <= 0.0))
                {
                    state = false; s.write(false);
                }
                if((state==false)&&(voltage > 0.0))
                {
                    state = true; s.write(true);
                }
            }
            SC_HAS_PROCESS(sca_diode_control);
            sca_diode_control(sc_core::sc_module_name): v("v"),i("i")
            {
                state = false;
                SC_METHOD(event_control); sensitive << v << i;
            }
            private:
            bool state;
        };
    }   // namespace sca_de
}       // namespace sca_eln
```

Fig. 10.4 SystemC diode control logic implementation

```
Error: SystemC-AMS: Initialization equation system failed in
sca_linear_solver_0: 1 the error is in the following net (max. 50):
circuit.R
circuit.L
circuit.S
circuit.E
circuit.D.S
circuit.D.Am
circuit.D.Vm
circuit.Am
circuit.C
circuit.G
circuit.Vm

The error is may  be near:
circuit.L  and
circuit.n2
```

Fig. 10.5 SystemC AMS error message during initialization

```
namespace sca_eln_mdl
{
    SC_MODULE(Control)
    {
        public:
        sc_core::sc_in<double> i_l;
        sc_core::sc_out<bool> g;

        void set_reference(double _reference){reference=_reference;}
        void set_p_threshold(double _threshold){p_threshold=_threshold;}
        void set_n_threshold(double _threshold){n_threshold=_threshold;}

        void event_control(void)
        {
            double current = i_l.read();

            if(((current - reference) >= p_threshold)&&(state==true))
            {
                state = false;  g.write(false);
            }
            if(((current - reference) <= n_threshold)&&(state==false))
            {
                state = true;  g.write(true);
            }
        }
        SC_HAS_PROCESS(Control);

        Control(sc_core::sc_module_name, double _reference = 1.0,
        double _p_threshold = 0.5, double _n_threshold = -0.5):
        i_l("i_l"),g("g"), reference(_reference),
        p_threshold(_p_threshold), n_threshold(_n_threshold)
        {
            state = true;
            g.initialize(true);
            SC_METHOD(event_control); sensitive << i_l;
        }

        private:
        bool    state;          // Relay state
        double  reference;      // Reference value for control
        double  p_threshold;    // Upper threshold
        double  n_threshold;    // Lower threshold
    };
}   // namespace sca_eln_mdl
```

Fig. 10.6 SystemC Buck converter controller implementation

The obtained simulation results are however not correct. As shown in Fig. 10.8, inductor current becomes zero after switching. Thereby, the diode does not remain turned ON and the controller closes the switch again to reach the desired inductor current. The reason for this behaviour is that SystemC AMS solver allows only one switching event at a given point in time. This restriction limits the type of circuits that may be simulated using SystemC AMS. The simulations results using multiple instantaneous switching are showed in Sect. 10.6 (Fig. 10.10).

```
Error: SystemC-AMS: Reinitialization failed in sca_linear_solver_0: 1
during reinitialization equation system at 21217 us (dt = 1e-06).
the error is in the following net (max. 50):
circuit.R
circuit.L
circuit.S
circuit.E
circuit.D.S
circuit.D.Am
circuit.D.Vm
circuit.Am
circuit.C
circuit.G
circuit.Vm

Parameters of the following modules changed for the current time step:
circuit.S changed to 0
circuit.D.S changed to 0

The error is may  be near:
circuit.D.Am
```

Fig. 10.7 SystemC AMS error message during simulation

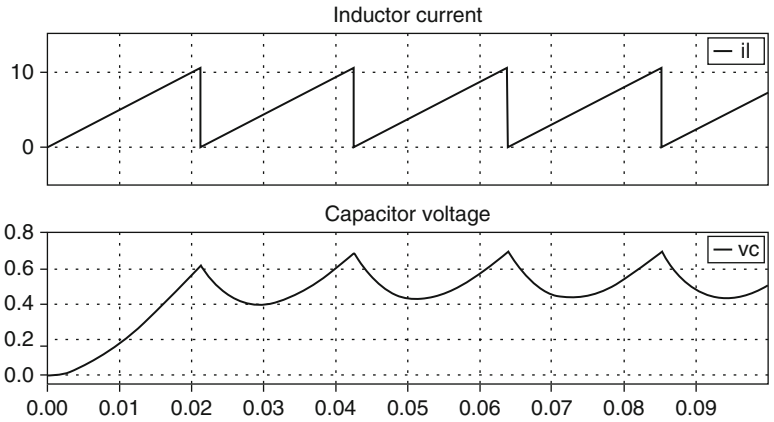


Fig. 10.8 Buck converter simulation results with SystemC AMS ELN

10.5 Modelling and Simulating Power Electronics Using Ideal Switches

The use of ideal instantaneous switches is often an appropriate behaviour abstraction to model power electronic semiconductors because the processes during switching are not important when power electronic circuits are simulated for system performance evaluation. An ideal or perfect switch has zero resistance when ON, zero admittance when OFF, and switches between both states in zero time. Representing

switches in this way leads to more robust and faster simulations. On the other hand, circuit state inconsistencies after topology change need to be properly handled during simulation.

10.5.1 Computational Advantages of Ideal Switching Modelling

1. Simple numerical integration algorithm:
   The numerical integration algorithm is applied during simulation to compute the electrical circuit behaviour after a circuit topology was defined. The method and parameters required to solve the associated differential equations depend mainly on current circuit characteristics. They may change after each switching event. Using linear electrical elements, an accurate representation of power electronic circuits is possible. Linear differential algebraic equations can be numerically solved using very simple integration techniques such as forward and backward Euler or the trapezoidal rule [20].
2. Short time switching process:
   Because ideal switches change their state in zero time, only two integration steps are necessary to handle the discontinuities at each switching event. The numerical integration algorithm is applied again to compute the system state after switching.
3. Reduced equation system:
   Nodes connecting ideal switches can be contracted into a single node when ideal switches are ON. Thus, the size of the network matrices and hence the computation time can be reduced.
4. Less solvability problems:
   Because ideal switches do not have very small or very large parameter values, the resulting circuit equation system is normally non-stiff which leads to stable simulations. Additional passive components (snubbers) to make the simulation converge are rarely necessary and therefore no parameter tuning is required.
5. More accurate models:
   Due to the large signal behaviour of power circuit models, avoiding additional resistances for solvability increases considerably the simulation accuracy. Furthermore, the additional parameters of non-ideal switches must be chosen according to the circuit being analysed. It requires a good understanding of the circuit's operation. Ideal switches eliminate the trade-off between accuracy and stability.
6. Faster simulations:
   Ideal switches do not affect system response. In opposite, the additional components of non-ideal switches introduced for simulation stability may considerably increase the simulation time, in particular, if their parameters are not properly chosen.

10.5.2 Computational Requirements for Ideal Switching Modelling

An ideal switch represents a short circuit when it is closed and an open circuit when it is open. Depending on the resulting topology after commutation, different circuit or state inconsistencies can be caused by:

- floating nodes
- short-circuited voltage sources
- open-circuited current sources
- short-circuited capacitors
- open-circuited inductors
- different initial conditions of circuit elements

If one or more inconsistencies are present in the circuit topology after switching, the resulting differential and algebraic equation system cannot be numerically solved. Using circuit analysis such network inconsistencies must then be found and removed from the system description matrices.

Additionally, the network response may be discontinuous at the switching instants, if inconsistencies are present. It means that voltage or current impulses may be generated by the state change. They can be also produced by inconsistent initial conditions [21]. Thus, impulsive currents or voltages need to be accurately recognized and appropriately handled for determining the correct topology after switching. Internally controlled switches may change their state if a current or voltage impulse is applied to them. As described in the related work, there are several methods to cope with voltage and current impulses in switched networks.

10.6 Electrical Piece-Wise-Linear Networks (EPN)

As presented in [19], the language architecture of SystemC AMS standard is defined in an extensible way. New models of computation can be defined and seamlessly integrated by using base classes for signals, ports and modules.

10.6.1 Syntax and Primitives

The proposed extension of SystemC AMS for modelling electrical circuits with ideal switches follows the same syntax as the currently available MoC ELN (Electrical Linear Networks). Thus, existing SystemC AMS models can be compiled and simulated with only minimal code modifications.

In order to support the specific features of ideal switching modelling with natural commutation, primitives for electrical elements were implemented using the

attributes and methods of the new solver class. We named our model of computation "Electrical Piece-wise-linear Networks" and assigned the namespace sca_epn to the new types and classes. All primitives are derived from the base class sca_module.

In order to utilize the built-in binding mechanism of SystemC, terminals are instances of a type derived from sca_port and nodes are instances of a type derived from sca_interface. Circuit elements are also interconnected by binding terminals to nodes. Primitives for modelling externally and internally controlled ideal switches (epn_switch and epn_diode) are provided to enable the novel language capabilities.

10.6.2 Network Equations Formulation

Similar to the current SystemC AMS implementation, we obtain circuit equations by applying the rules of the MNA. The nodal equation set-up is carried out according to the following expressions:

$$[Y_n B_b]\,[V_n] = [J_n] \tag{10.1}$$

$$[B_n Z_b]\,[I_b] = [E_b] \tag{10.2}$$

where Y_n is the nodal admittance matrix, Z_b is the branch impedance matrix, B_b is the branch incidence matrix, B_n is the nodal loopset matrix, J_n represents the equivalent nodal current sources, E_b represents the equivalent branch voltage sources, V_n is the nodal voltage vector and I_b is the branch current vector.

Matrix stamps for this equation partition are formulated as described in [20]. This leads to less equations than the current SystemC AMS implementation and is more appropriate for power circuit simulation consisting of linear elements.

During the SystemC AMS initialization phase, the matrix_stamp function is called for each instantiated circuit element. The previously defined matrices, Y_n, B_b, B_n and Z_b, are then created and used to compute the system matrix by applying the numerical integration algorithm as described in [20]. Linear multistep methods (LMS) are currently provided.

The value of the system matrix coefficients depends on the state of the internally and externally controlled switches. The resulting circuit topology is then analysed considering the criteria specified in the previous chapter.

10.6.3 Topology Analysis of Switched Electrical Networks

To carry out an efficient topology analysis of switched electrical networks we extended graph theory representation methods for the detection of network inconsistencies. A network graph considers the network elements as terminal components. It describes the connection between network elements, capturing the properties of

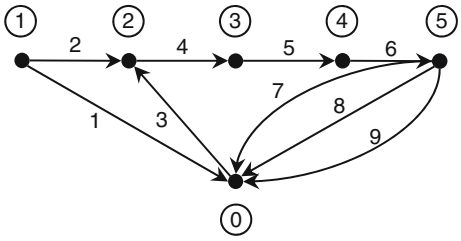


Fig. 10.9 Graph representation of the Buck converter circuit

Table 10.1 Buck converter representation using a table

Edge	1	2	3	4	5	6	7	8	9
Type	E	S	D	L	R	A_m	C	G	V_m
N_p	1	1	0	2	3	4	5	5	5
N_n	0	2	2	3	4	5	0	0	0
Z_i	0	1	2	0	0	0	0	0	0

the network in a natural way. Network graphs can be efficiently implemented on the computer in form of a simple table.

1. Network representation using a directed graph
   For the construction of the directed graph network representation, each two-terminal element of the network is numbered and replaced in the circuit by a line called edge. An orientation corresponding with the assigned current flow direction is associated with each edge in the graph. For passive elements, the node from which the current flows is the positive terminal. For current sources, the direction of the current is defined by its symbol. For voltage sources, the direction of the current flow is from the positive to the negative terminal. The vertex of the graph represents the nodes of the network. They are numbered assigning zero to the ground. The node numbers in the graph are placed within circles to distinguish them from the edge numbers. Figure 10.9 shows the graph representation of the Buck converter circuit.
2. Network representation using a table
   For the construction of table representing an electrical network, each two-terminal element of the network adds a new column to the table. The edge, the type, the positive and the negative node of the each network element are described in the corresponding table row. For switched network analysis we add a row containing the state variable number of switching elements.

Table 10.1 buck converter circuit. N_p, N_n and Z_i represent the positive node number, the negative node number and the discrete state variable number respectively.

In order to handle two ports elements by a single graph, the input and the output terminals are represented as separated edges (columns in the table representation form).

Table 10.2 Reduced Buck converter representation using a table

Edge	1	2	3	4-5-6	7-8-9
Type	E	S	D	L-R-A_m	C-G-V_m
N_p	1	1	0	2	5
N_n	0	2	2	5	0
Z_i	0	1	2	0	0

10.6.3.1 Topology Analysis Using a Network Graph

The graph representation of a switched electrical network can be utilized to reduce the number of equations and predict inconsistent equations. The following information about the network nodes and branches can be obtained:

- connected elements which can be reduced to a single element
- floating nodes and its switching dependencies
- floating network elements
- complementary network branches (i.e. diode bridge)

The following steps are carried out during the graph based topology analysis of a switched network:

1. Group serial connected impedances (R-L-A_m elements which may be represented as single impedance or current branch).
2. Group parallel connected admittances (G-C-V_m elements which may be represented as single admittance).
3. Identify floating nodes if all switches are open.
4. Identify floating branches if all switches are open.

A floating node is a node to which only one element is connected. The current through a branch connected to a floating node is zero. A floating branch is a branch terminated with floating nodes. It is not possible to compute the voltage across its nodes.

Applying the steps 1–2, a reduced circuit table is obtained (Table 10.2) for the Buck converter circuit. The elements L, R and A_m have the same current and can be grouped into a single edge. The parallel admittance branches G-C have the same voltage applied to its nodes and can also be grouped into a single edge. The output voltage V_m is the same as the voltage across G-C.

If all switches are open (step 3), there two floating nodes in the circuit (node 1 and node 2). Because all circuit branches are connected to ground when all switches are open, does not exist any floating branch which need to be eliminated from the network equation system (step 4).

10.6.3.2 Voltage and Current Graphs Analysis

An edge in an electrical network graph simultaneously represents the current through the element and the voltage across the network element. For some elements of the network, one of the constitutive variables may be zero.

For example, the current through an ideal switch or the voltage across its terminal is zero when it is open or closed respectively (complementary behaviour). For some other elements, one of the constitutive variables is not necessary for the solution of the resulting equation system and it is also not of interest, such as the current through a voltage source and the voltage across the terminal of a current source. Using separated graphs to represent the network voltages and currents, this redundancy can be eliminated (see [20] for more details).

Extended current and voltage graphs can be utilized for the prediction of network topologies producing impulses.

1. Topology analysis using the current graph (I-graph)
   The current graph can be utilized to get the following information about the switched electrical network:
   - open circuited branches which may generate voltage impulses (branches containing current sources and inductors)
   - switching dependencies for such open circuited branches
   - voltage impulses acting on internally controlled switches
2. Topology analysis using the voltage graph (V-graph)

The voltage graph can be utilized to get the following information about the switched electrical network:

- short circuited elements branches which may generate current impulses (voltage sources and capacitors)
- switching dependencies for such short circuited elements
- current impulses acting on internally controlled switches

10.6.3.3 Current Graph Topology Analysis Steps

The following steps are carried out during the current graph based topology analysis of a switched network:

1. Collapse the nodes of the network elements that are not of interest for network equation formulation (voltage sources E and sinks V_m).
2. Collapse the nodes of the network elements that are not of interest for impulse analysis (only switches and branches containing inductors remain)
3. Identify open circuited branches containing current sources and inductors (branches which generate voltage impulses).
4. Identify the switching state dependencies for branches generating voltage impulses.
5. Identify voltage impulses acting on internally controlled switches and their polarity.

Applying the steps 1–2, a reduced circuit table is obtained (Table 10.3) for the Buck converter circuit. If all switches are open, the branch 4-5-6 is not closed.

Table 10.3 Reduced Buck converter representation table for current analysis

Edge	2	3	4-5-6
Type	S	D	L-R-A_m
N_p	0	0	2
N_n	2	2	0
Z_i	1	2	0

Table 10.4 Reduced Buck converter representation table for voltage analysis

Edge	1	2	3	4-5	7-8-9
Type	E	S	D	L-R	C-G-V_m
N_p	0	0	0	0	5
N_n	0	0	0	5	0
Z_i	0	1	2	0	0

Using the reduced table, it is not difficult to find the states producing voltage impulses. In this case, if the switch S (state z_1) and the diode D (state z_2) are open, a voltage may impulse occur (the inductor current must be greater than zero). Additionally, we can find out that the produced voltage impulse, when the switches are open, is applied to the diode D with positive polarity.

10.6.3.4 Voltage Graph Topology Analysis Steps

The following steps are carried out during the voltage graph based topology analysis of a switched network:

1. Collapse the nodes of the network elements that are not of interest (current sources and sinks).
2. Identify short circuited voltage sources and capacitors if all switches are closed.
3. Identify the switching state dependencies for branches generating current impulses.
4. Identify current impulses acting on internally controlled switches and their polarity.

Applying the step 1, a reduced circuit table is obtained (Table 10.4). The voltage through A_m is zero, and its nodes can be collapsed. After collapsing all switch nodes, the current impulses generated by short circuited voltage sources and capacitors can be founded (step 2). In our example, the battery E is short circuited when the switch S and the diode D are simultaneously closed (step 3). The current impulse is then applied to the diode D and the corresponding current flow direction is determined by the polarity of the battery (step 4).

10.6.4 Solver Implementation for Switches Networks

The topology analysis described in the previous section is carried out by the solver before the simulation starts. It provides the necessary information for the computation of the current topology equations. If circuit inconsistencies are detected during the equation setup; the rows and columns associated with non-valid branches and floating nodes are shifted to the outside of the resulting system matrix. The solver variables indicating the number of system nodes and branches are respectively modified.

The system matrix creation and analysis is repeated during circuit simulation, when switching conditions are reached or circuit equations modified. In order to improve simulation performance, the system matrix is factorized using LU decomposition. Additionally, the computed topologies can be stored. The number of stored topologies can be limited by the user.

As shown in Fig. 10.10, the solver algorithm calls a list of pre-solve methods at the start of the integration step. They are dynamically registered by the circuit elements and managed by the solver. This SystemC AMS functionality is utilized by our ideal switches to read SystemC ports and update circuit topology if necessary. If state changes are reported, the system matrix is computed again. The pre-solve methods are called again if inconsistencies were detected. Forward and backward substitution is applied at each integration step to obtain the system response.

After one integration step is carried out, the solver calls the post-solve methods. The diode class exploits this mechanism to evaluate switching conditions. It sets a solver flag if their threshold values are reached. The solver creates the system matrix and repeats the integration step, if a natural switch condition was reported. Because ideal diodes modify their state instantaneously, this loop needs to be executed until no more switching conditions are detected. If during the switching process a topology occurs twice, the system is not stable and the simulation is aborted.

10.6.5 Time Step Control

In order to keep the errors caused by the numerical integration small as well as to synchronize continuous and discrete time parts, analogue simulators carry out step size control during simulation. The SystemC kernel employs an entirely different approach to control time advance. It uses events that trigger processes. The solver time control is implemented in SystemC AMS using a spawn process.

Due to the large signal behaviour of power electronic circuits, a very short integration step is often required to minimize numerical integration errors. This can notably decrease the simulation performance. As shown in Fig. 10.10, a step refinement loop was incorporated into the solver algorithm to improve the simulation time and obtain accurate results. A solver variable controlling the loop can be modified by the primitive pre- and post-solve methods. In a first approach this

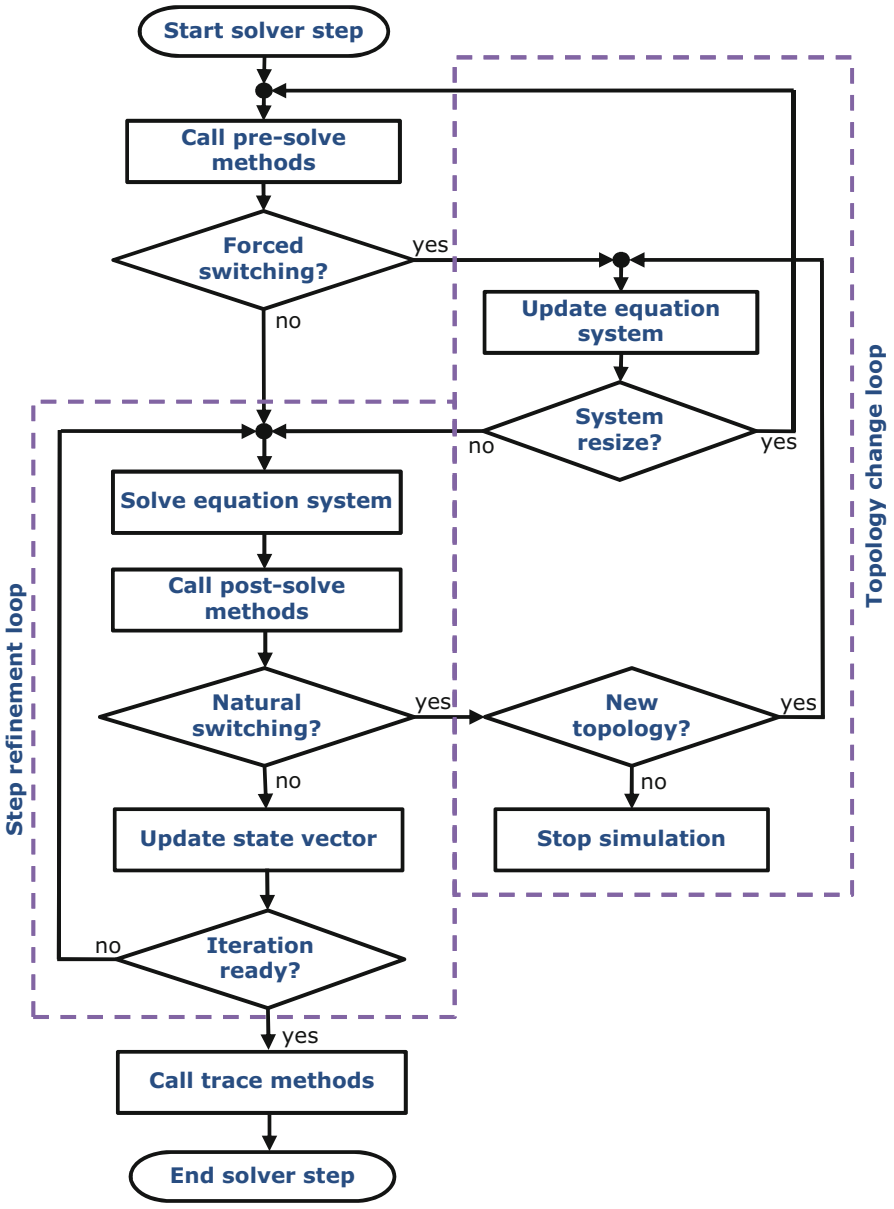


Fig. 10.10 Solver step refinement and topology change loops

simulation parameter is adjusted by the user by calling a solver interface function in the SystemC module constructor.

To split solver interface from step refinement and topology change loops, trace methods were defined and implemented for current and voltage sinks.

10.6.6 Electrical Circuit Integration in Simulink

The control algorithm of the power system is normally implemented as Simulink model. In order to validate the power circuit functionality in Simulink, a set of libraries were created. They enable the smooth integration of electrical circuits, implemented in SystemC, into Simulink models by embedding user code. The SystemC module registration and port binding process is carried out by the Simulink block frame when the model initialization function is called. During simulation input and output signals are writing to and reading from global variables assigned to the SystemC ports. The input and output port order follows SystemC declaration order.

10.7 Experimental Results

10.7.1 Buck Converter Simulation

The buck converter circuit, proposed in Sect. 10.4 to illustrate the current SystemC limitations by simulating internally controlled switched networks, was modelled and simulated using ideal instantaneous switches. As shown in Fig. 10.11, the inductor current does not present discontinuities.

This demonstrates that our approach enables multiple instantaneous switching.

At time 0.021 s, the switch S is open and the impulse generated by the inductor turns the diode ON. At time 0.091 s the switch S is closed and the negative voltage applied to the diode turn it OFF. The short circuit caused by the diode, when the switch is closed is properly handled by our algorithm.

In Table 10.5, the execution time of this model till the switch is opened at 0.021 s is compared for different approaches and data sampling frequency. The EPN MoC compute 10 ms faster, when the solver step size is increased from 1 to 10 μs and the number of step iterations set to 10. Simulink simulates faster when the model is only loaded on memory.

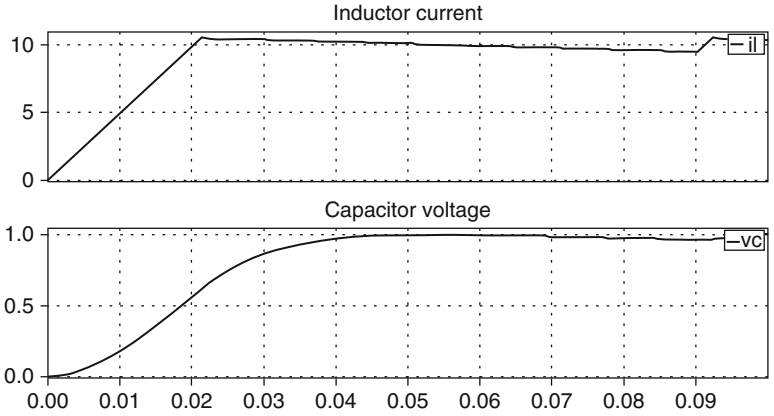


Fig. 10.11 Buck converter simulation results using ideal instantaneous switches

Table 10.5 Buck converter simulation execution time

	Simulation execution time		
Data sampling	SystemC ELN	SystemC EPN	Simulik/PLECS
10 µs	60 ms	40 ms | 50 ms	62 ms | 74 ms
100 µs	50 ms	40 ms | 50 ms	60 ms | 69 ms

10.7.2 High-Voltage Power Converter Simulation

In order to evaluate the practical application of the described SystemC AMS extension for power electronic modelling, a high-voltage power converter used in medical machines has been modelled and simulated. Figure 10.12 shows a block diagram representation of power electronic system. Due to the large number of switches (4 externally controlled and 16 internally controlled), the high frequency operation (10 ns), as well as the very large output signal range (30–120 kV) this switched electrical network provides the required complexity to validate the proposed methodology and its software implementation.

The digital controller was implemented using discrete event modules. It operates in three levels (ON, OFF and CONTROL) to achieve a rapid response. Many signals are monitored to control the output voltage. As shown in Fig. 10.13, there is a strong analogue and digital interaction.

Although circuit parameter values comprise a very large range, solvability problems were only encountered for one topology. The modelling task was considerably simplified by using ideal diode primitives. As expected, the small integration step leads to large simulation times. Simulation results were similar to those obtained with PLECS by using fixed time step.

The simulation accuracy allows an adequately analysis of signal harmonics.

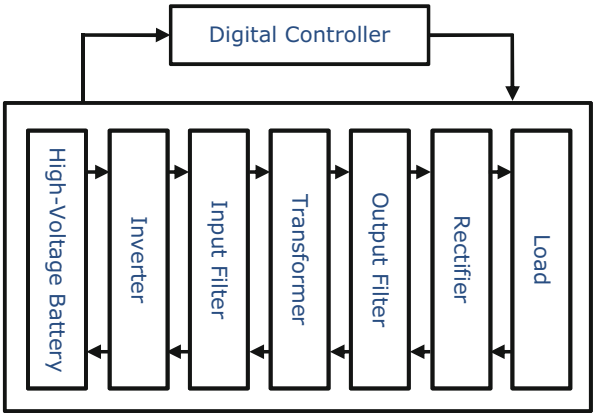


Fig. 10.12 Block diagram of the high-voltage power converter

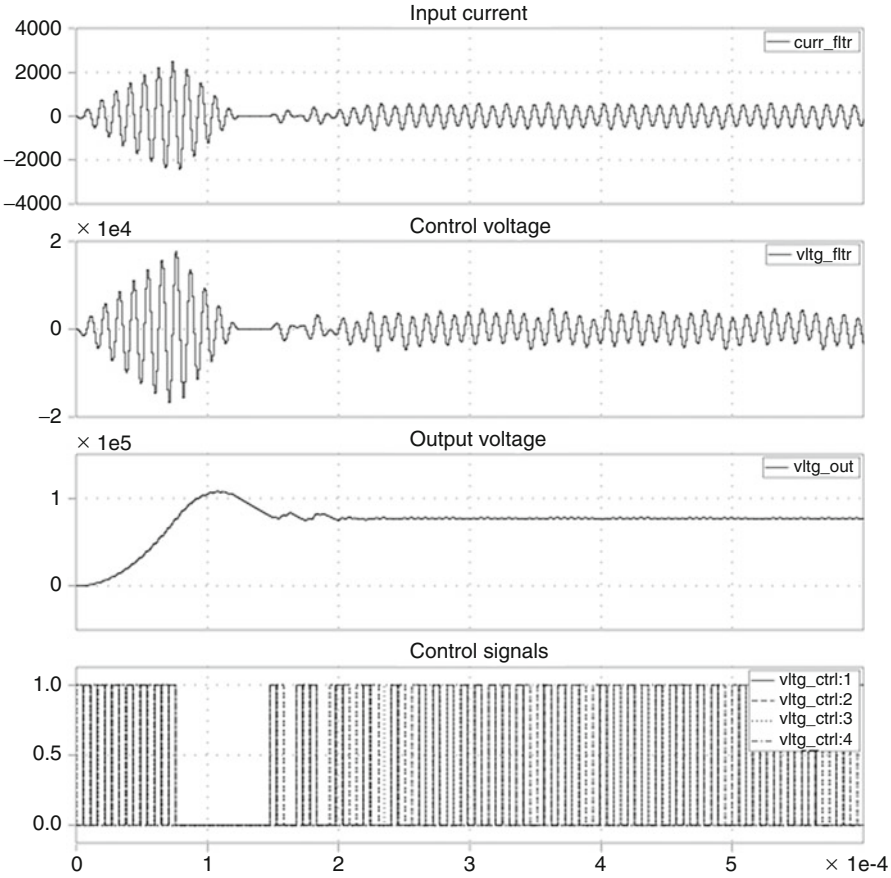


Fig. 10.13 Power converter signals

10.8 Conclusion

In this paper we proposed an extension of SystemC AMS for modelling and simulation of power electronic circuits. It was integrated into the existing language architecture. Our proposed implementation hides switching control details from the models, resulting in a simple and more efficient modelling approach of electrical piece-wise-linear networks. Experiments show that complex internally controlled electrical switched networks are robust and accurately simulated in SystemC. We are encouraged by the obtained results to continue with the development of modelling abstractions for efficient power circuit simulations in SystemC.

In the next step we will improve solver efficiency to carry out large system level simulations. In the future, we want to investigate other network abstractions taking advantage of circuit properties and the inherent repetitive operation mode of power systems.

Acknowledgment The authors would like to thank Philips Research for the contribution with power system models and the German Federal Ministry of Education and Research (BMBF) for financial support of this work within the project POWERBLOCK+ (grant number 16M3200F).

References

1. Al-Junaid, H., Kazmierski, T.: An introduction to modeling embedded analog/mixed-signal systems using SystemC AMS extensions. In: Proceedings of the International Symposium on Circuits and Systems (ISCAS'04), vol. 5, pp. 281–284, May 2004
2. Al-Junaid, H., Kazmierski, T.: An extension to SystemC to allow modelling of analogue and mixed signal systems at different abstraction levels. Conference or Workshop Item (Speech), September 2004
3. Al-Junaid, H., Kazmierski, T.: Analogue and mixed-signal extension to SystemC. In: IEE Proceedings of Circuits, Devices and Systems, pp. 682–690, December 2005
4. Allmeling, J., Hammer, W.: PLECS – Piece-wise Linear Electrical Circuit Simulation for Simulink. In: Proceedings of the IEEE International Conference on Power Electronics and Drive Systems (PEDS '99), vol. 1, pp. 355–360, July 1999
5. Allmeling, J., Hammer, W.: Simulating power electronic systems using ideal instantaneous switches. In: Proceedings of the International Conference on Power Electronics, Intelligent Motion, Power Quality (PCIM Europe '04), vol. 2 pp. 585–590, May 2004
6. Bedrosian, D., Vlach, J.: Time-domain analysis of networks with internally controlled switches. In: Proceedings of the IEEE International Symposium on Circuits and Systems, vol. 2, pp. 846–849, June 1991
7. Bedrosian, D., Vlach, J.: Time-domain analysis of networks with internally controlled switches. IEEE Trans. Circuits Syst. 1 Fundam. Theory Appl. **39**(3) (1992)
8. Bozin, A.: Electrical power systems modeling and simulation using SIMULINK. In: Proceedings of the IEE Colloquium on the Use of Systems Analysis and Modelling Tools: Experiences and Applications, pp. 10/1–10/8, March 1998
9. Einwich, K.: SystemC AMS Extensions–The Language. Tutotial, September 2010
10. Grimm, C., Meise, C., Oehler, P., Waldschmidt, K., Fey, F.: AnalogSL: A library for modeling analog power drivers with C++. In: Proceedings of the Forum on Specification and Design Languages (FDL '01), September 2001

11. Grimm, C., Barnasconi, M., Vachoux, A., Einwich, K.: An introduction to modeling embedded analog/mixed-signal systems using SystemC AMS extensions. Open SystemC Initiative, June 2008
12. Massarini, A., Reggiani, U.: Computer-aided time-domain large-signal analysis with network switches. In: Proceedings of the International Symposium on Industrial Electronics (ISIE '96), vol. 2, pp. 567–572, June 1996
13. Massarini, A., Kazmierczuk, M.K.: A new representation of Dirac impulses in time-domain computer analysis of networks with ideal switches. In: Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS '96.), vol. 1, pp. 565–568, May 1996
14. Massarini, A., Reggiani, U., Kazimierczuk, M.K.: Analysis of networks with ideal switches by state equations. IEEE Trans. Circuits Syst. I Fundam. Theory Appl. **44,** 692–697 (1997)
15. Plexim GmbH: An introduction to solvers. In: Vehicle Power and Propulsion Conference (VPPC), 2011 IEEE International, pp. 1–132, September 2011
16. Plexim GmbH: PLECS User Manual. http://www.plexim.com/files/plecsmanual.pdf
17. Reuther, C., Einwich, K.: A SystemC AMS extension for controlled modules and dynamic step sizes. In: Proceedings of the Forum on Specification and Design Languages (FDL '12), pp. 90–97, September 2012
18. Uhle T., Einwich, K.: A SystemC AMS extension for the simulation of non-linear circuits. SOC Conference (SOCC), 2010 IEEE International, pp. 193–198, September 2010
19. Vachoux, A., Grimm, C., Einwich, K.: Towards analog and mixedsignal SOC design with systemC-AMS. In: Proceedings of the International Conference on Field-Programmable Technology, 2004, pp. 97–102, January 2004.
20. Vlach, J., Singhal, K.: Computer Methods for Circuit Analysis and Design, 2nd edn. Van Nostrand Reinhold, New York (1993)
21. Vlach, J., Opal, A., Wojciechowski, J.: Simulation of networks with inconsistent initial conditions. In: Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS '93), vol. 3, pp. 1627–1630, May 1993